Constraint Satisfaction Approaches to Bus Driver Scheduling

by

Suniel David Curtis

Submitted in accordance with the requirements
for the degree of Doctor of Philosophy.

The University of Leeds
School of Computer Studies

February 2000

The candidate confirms that the work submitted is his own and the appropriate credit has been given where reference has been made to the work of others.

Abstract

The bus driver scheduling problem consists of assigning bus work to drivers so that all the bus work is covered and a combination of the number of drivers and associated costs is minimised. Restrictions imposed by logistic, legal and union agreements complicate the problem.

Successful present day systems for computerised driver scheduling often use mathematical programming combined with heuristics. Purely heuristic approaches have found it very difficult to produce efficient driver schedules for large scheduling problems. Furthermore, some of these approaches may not be easily adaptable to different conditions. This thesis presents two new ways of using constraint satisfaction to form driver schedules. The two methods differ in their approach, one being a systematic constraint programming approach and the other being an adaptation of a local search method called GENET.

The constraint programming approach uses a similar approach to mathematical programming systems in selecting the schedule from a large number of possible shifts, to allow adaptation to different regulations. In particular, a set partitioning formulation is used. It then makes use of the structure of the problem and the relaxed linear programming solution to the problem in producing a schedule. The GENET system has been adapted to cope with minimising the numbers of drivers in a schedule and with the memory problems caused by the huge number of constraints involved in the set partitioning model.

The constraint programming approach has been shown to solve successfully several small scheduling problems from different companies using varying regulations. Local search procedures have hitherto not had great success on driver scheduling problems. GENET has been adapted to solve some of the small schedules from its initial state where it could not solve any. Features of the adaptation may be of interest to researchers using GENET on similar problems.

Acknowledgements

I would like to thank my supervisors, Dr. B. M. Smith and Professor A. Wren, for guidance. Further, my supervisors (again) and the friends/family who have given their support and encouragement that is important to me as a dyslexic, you know who you are thank you.

Declarations

Some parts of the work presented in this thesis have been published in the following articles:

S. D. Curtis, B. M. Smith, and A. Wren, "Forming Bus Driver Scheduling using Constraint Programming", *Practical Application of Constraint Technologies and Logic Programming PACLP99*, (1999) 239–254.

S. D. Curtis, B. M. Smith, and A. Wren, "Constructing Driver Schedules using Iterative Repair", *Practical Application of Constraint Technologies and Logic Programming PACLP2000*, (2000) 59–78.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In present day industry competition is so fierce that cutting costs is paramount and improved schedules and timetables can make huge monetary savings. This is true of driver scheduling which is an important real world problem as crew costs account for a high proportion of total expenditure in most transport companies. In the UK now that the transportation industry is privatized profits are important and with consumer concern about high fares, making the running of bus and trains efficient is the best way to maximise profit. The precursor to the driver scheduling problem is the bus scheduling problem where routes need to be worked out and vehicles assigned to them. Once this is done the bus driver scheduling problem involves finding the most efficient way of providing drivers for the given set of bus movements, including dead running (journeys with no passengers). These two problems tend to be kept separate due to both problems being individually hard. If they were combined the ensuing problem would surpass current computer scheduling methods run on standard machines used by transport companies.

There are several restrictions on efficient provision of driver schedules, imposed by legal

and logistical considerations as well as trade union agreements. For example, a driver may only legally drive a certain number of consecutive hours. The criterion is usually that the schedule should have the minimum number of shifts and lowest total hours of work. The total hours of work is normally a secondary consideration and because of this it is disregarded in the new method implemented in this thesis.

1.1 Computerised driver scheduling

Early computerised methods for driver scheduling were purely heuristic and often needed large amounts of manual intervention. As methods and computer power improved mathematical programming started to be used. In the present day there are some very good systems, for example TRACS II [37, 66, 125], which can provide efficient schedules for very large problems. Despite this the modern systems cannot be seen as black boxes that produce working schedules. TRACS II has been adapted for several bus and rail companies and has through long development and experience working with these companies reached a level of generality so it can fit with many companies' requirements. However, even after this has been done parameters must sometimes be manipulated to produce driver schedules for different bus schedules. Such manipulation is frustrating and perhaps obscure to schedulers who have no knowledge of mathematical programming. This brings us to several areas where improvements can be made. Firstly, the driver scheduling problem is still open, in that optimal results cannot be ensured by current methods for any but the most trivial instances. Secondly, flexibility can be improved; although great strides have been made with the mathematical approach there are some aspects of scheduling that are hard to incorporate in a linear programming model. Thirdly, the present mathematical approaches are hard to explain to people not versed in science disciplines and this is not only, as stated above, a problem in producing individual schedules, it is a hindrance in mutual development of systems between researchers in universities and scheduling groups within companies.

1.2 Thesis overview

We have already stated how important the problem is and that there is room for improvement. In the previous section three areas were highlighted as areas for development. The last two, flexibility of the model and understanding of the user, are the ones that this thesis is concerned with. It is felt that the expressive qualities of the modelling language of constraint satisfaction will be of use in these areas and therefore constraint satisfaction approaches are investigated in this thesis.

This thesis will explore two new approaches for producing bus driver schedules. One is a systematic approach using a constraint programming method and the other is a local search method called GENET [121, 110]. The thesis not only provides new research in the area of bus driver scheduling but allows a comparison of three of the popular fields of research for solving combinatorial problems: mathematical programming, constraint programming and local search. They will be compared only on one type of problem, driver scheduling, but each technique will be investigated in depth.

The following summarises the contents and reason for each chapter.

Chapter 1: Gives motivation for the new research and gives an overview of the thesis.

Chapter 2: Introduces concepts of constraint satisfaction and constraint programming. This concentrates on the methods used in the thesis and a discussion on arguably the most important issue in constraint satisfaction, modelling.

Chapter 3: Gives a history of the build up to the local search method GENET which is investigated in Chapter 6. It also gives a brief overview of other local search methods for constraint satisfaction problems.

Chapter 4: Gives a brief history of driver scheduling. It gives reasons why the problem is still open, in that optimal results cannot be ensured by current methods for any but the most trivial instances.

Chapter 5: Details the constraint programming approach developed for producing driver schedules. It shows that the program is successful on several small bus driver scheduling problems and shows potential for marked improvement.

Chapter 6: Details the adaptation of the local search method, GENET for constructing driver schedules. It gives promising results for several bus driver scheduling problems.

Chapter 7: Discussion of the existing mathematical approach and the two new approaches. This includes thoughts on their potential and possible further work.

Chapter 2

Constraint Programming

2.1 Introduction

Constraint satisfaction approaches for solving industrial problems are becoming more widely used because they provide a good method of tackling large problems in a flexible and adaptable way. Constraint satisfaction provides a powerful and easy system for modelling restrictions and using these restrictions to search for a solution.

There are several definitions that will be presented here to provide a background to the work in this thesis (see [106] for these and further definitions).

A *domain* of a variable is the set of possible values that the variable can take. A variable x_i will have a domain D_i. In this thesis we will only have variables with finite domains.

An *assignment* is a binding of a variable (u) to a value (v) to form a *label* $< u, v >$. The *label* is the variable-value pairing.

A *compound label* is a simultaneous assignment of variables to values. A *k-compound label* is an assignment of k labels simultaneously and can be represented as $(< u_1, v_1 >< u_2, v_2 > \ldots < u_k, v_k >)$.

A *constraint* restricts the values that variables can be assigned to simultaneously. Formally a constraint can be defined as a set of legal compound labels, although for efficiency and expressive reasons constraints can be defined in many ways, such as equations, matrices, functions, etc. The number of variables that the constraint acts on is called the *arity* of the constraint. If it acts only on 2 variables it is called a binary constraint. A binary CSP is a CSP where all the constraints are binary or unary. In this thesis we will be using mainly binary CSPs. A *nogood* is a constraint on a pair of labels which states that both cannot simultaneously be chosen.

A *solution* to a CSP in this thesis means an assignment of a value to every *variable*. In a *feasible* solution all the constraints are satisfied, formally a member of the set of compound labels of each constraint exists in the solution. In an *infeasible* solution constraints are broken (not satisfied).

A constraint satisfaction problem (CSP) consists of a set of variables (Z), a function (D) which maps every member x_i of Z to its domain D_i and a set of constraints (C), a set of all legal sets of compound labels. So a CSP is represented as the triple (Z, D, C).

A binary CSP can be represented as a graph, where the nodes of the graph correspond to the variables and the edges or arcs represent binary constraints between variables. A constraint is bi-directional and so can be represented as an undirected edge. However, it is often useful to represent a constraint as two arcs, one for each direction of the constraint. So two nodes, x and y can be connected by a constraint represented as the arcs, (x,y) and (y,x). We define an arc (x,y) to be *arc consistent* if and only if for every value a in the domain of x there exists a value in y that is compatible with the label $< x, a >$. We can propagate the effect of a constraint by removing values that do not satisfy this arc consistent property for the arcs representing the constraint. This is called constraint propagation. An example of this process is shown in Figure 2.1. The constraint is a simple

greater than (>) constraint. Figure 2.1 (a) shows the original states of the domains of the variables before constraint propagation. Then (b) shows the arc (x,y) being made arc consistent. Finally both arcs, (x,y) and (y,x) are arc consistent in (c).

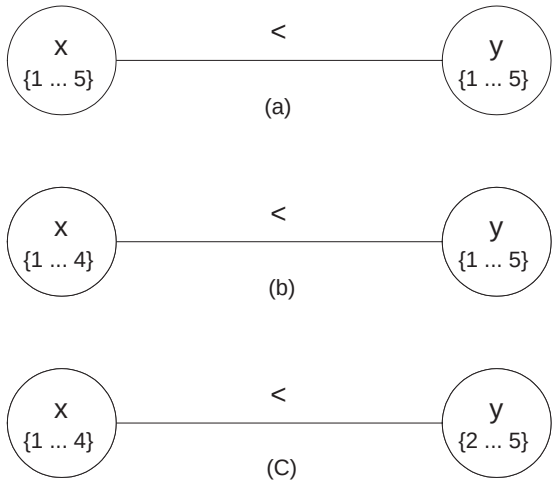

Figure 2.1: Making the constraint arc consistent

Using constraint programming tools can greatly increase the ease of programming CP algorithms. It also forms a base for sharing and comparing code and algorithms in the research community. These CP tools provide the user with implementations of standard processes involved in constraint programming, such as arc consistency. They also define a structure for the modelling problems and development of algorithms. The one used in part of this thesis is a C++ library called ILOG Solver [85]. There are however, several other tools such as ECLiPSe [118] and Chip [52] both based on Prolog.

A standard example of a problem that has been represented as a CSP is the n-queens problem. The problem is to put a number (n) of queens on a $n \times n$ chessboard without attacking any others, so no queen can be in the same row, column or diagonal as another. A simple way of representing the problem is to have the queens as the variables. So each queen can take any place in the $n \times n$ chessboard and the domain of each variable is all the squares of the board. There are then constraints to specify that no two queens are in the same row, column or diagonal. This is actually a poor representation and Section 2.7 on modelling shows other ways of representing it.

2.2 The basics of systematic complete search

The simplest form of systematic complete search using constraints is called BT [47]. The basic form of this search consists of the following. The variables are ordered arbitrarily. Then working through the variables in this order, for each variable assign to it the first value in its domain. This assignment is checked to make sure it is compatible with all the previously assigned variables. If it is not compatible a new assignment is tried and the current value is temporarily removed from the domain of the current variable. If no label associated with the current variable is compatible the algorithm *backtracks* to the previously assigned variable and a new value is tried for it. This case is called a failure or a fail. The search terminates if a solution is found or there is nowhere to backtrack to after a fail, which signifies there is no feasible solution for the problem. This termination property makes BT a complete search; if there is a feasible solution given time it will find it and if there is no feasible solution it will prove there is none. BT forms the basis of several search algorithms described in this chapter and this makes them all complete searches. Figure 2.2 shows the BT procedure. In the BT algorithm no advantage is taken of any constraint propagation. An improvement of this procedure is FC [51] it is the same as BT except in the way it performs consistency checks. Every time an assignment is made the values inconsistent with all the labels chosen are removed from the domains of all unassigned variables. The choice *fails* and the algorithm backtracks if any variable's domain becomes empty. There is no need to check an assignment's compatibility with earlier assignments because if it was incompatible it would have been removed at the

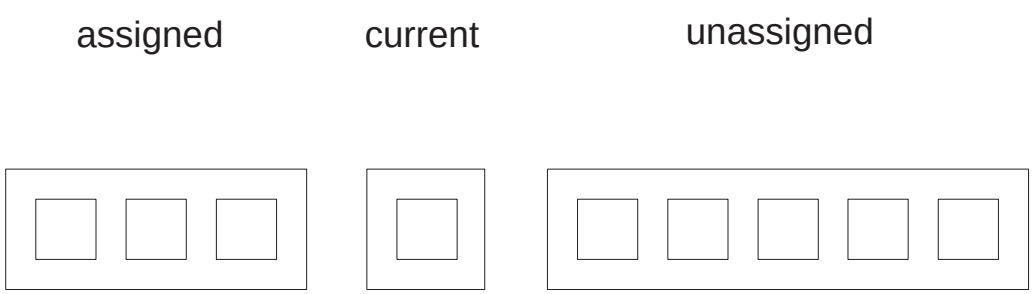

Figure 2.2: Simple form of search

time the previous assignment was made. However, it may occur that two unassigned variables have values compatible with all the assigned variables but not with each other. This can be resolved by adding an arc-consistency algorithm which checks for this type of inconsistency at every assignment. This is called *arc-consistency lookahead* [107] or maintaining arc-consistency (MAC) [90] where both forward checking and arc-consistency are used. Therefore, not only are the domains of the unassigned variables made compatible with chosen labels, they are also compatible with each other. Ways of maintaining arc-consistency will be discussed below.

There has been debate on the best arc-consistency algorithm. There has even been debate as to the usefulness of maintaining full arc-consistency during search [51, 90]. This is because the more times that consistency is checked for the greater the overheads on each assignment, as more checks need to be done. However, it is hoped that the more extensive checks will reduce the amount of backtracking and fruitless searching. Early work by Haralick and Elliot [51] suggested that only a limited amount of consistency checking should be used. However, later work by Sabin and Freuder [90] suggests that it is useful to apply full arc-consistency during the search. The difference in view might be that Sabin and Freuder focused on harder random problems than Haralick and Elliot. Further, AC algorithms have improved over time as described below. To ascertain what level of consistency to apply depends on the problem being solved and is still an open question.

2.3 Implementations of AC and MAC/AC lookahead

AC can be established as a pre-processing stage and as we have noted above can also be incorporated into search. In this section we will describe the details of several of the algorithms for establishing AC and then how these algorithms can be used in search.

Algorithms for establishing AC have been developed over time. The first three variations are described in [76]. They are all similar and the final one of this series, AC-3 informally consists of queuing all the binary constraints and then going through this queue propagat-

ing the effect of each constraint. As the constraints are propagated, the constraints that are associated with the variables that have their domains reduced are added to the end of the queue. Therefore, the queue will only become empty when no more domains are reduced by constraint propagation. More formally, when we say a constraint is added to the queue we mean only one of the arcs representing the constraint is added. Therefore, checking the arc (i,j) means that we will check that the values in the domain of variable i are consistent with those in the domain of j but not vice versa, so the actual additions to the queue works in the following way, if the domain of i changes arcs (i,j) for all existing j are added to the queue.

After AC-3 the next important development (AC-4 in [79]) in the AC algorithms was the idea that values support other values and when these supporting values are removed the supported value should be removed. This process saves consistency checks but requires additional memory because it stores all the supporting values and a counter that is incrementally decreased as these supporting values are removed. It is shown that AC-3 has a worse case time complexity of $0(d^3e)$ where AC-4 has $0(d^2e)$; d is the size of the largest domain and e is the total number of constraints [77, 79]. Whereas the space complexity of AC-3 is $0(e+nd)$, where n is the number of variables and AC-4 is larger, $O(d^2e)$. Further, it has been shown that in the average time complexity of AC-4 is close to its worse case and AC-3 often runs faster [119]. AC-5 [28] differs from the previous AC algorithms by giving only a framework for applying AC. It allows the consistency checks to be done differently by different constraint types. This allows the user to provide the most efficient algorithm to take advantage of a particular class of constraints. It does this by altering the queue that is used in AC-3. Instead of just queuing the constraints (e.g. C(i,j)) it also includes the values Δ that have been removed from the variable associated with the constraint that we are removing values from (i). Deville and Hentenryck [28] give examples of how this can be used to improve the efficiency of some types of constraints. For these AC-5 is a O(ed) algorithm. AC-5 allows users to provide constraint types and we will see how Solver allows this below. AC-6 [5] improves on AC-4 by reducing the space complexity down to $0(de)$ while maintaining the time complexity of $0(d^2e)$. It does this by storing

supporting values as AC-4 does, but instead of storing all the supporting values, it only stores one per constraint. If this value is then removed it looks for another. There has been several improvements on AC-6 and these culminate in AC-7 [6]. AC-7 extends the process by using inference. For example, when establishing that value a in the domain of u supports the value b in the domain of v we can infer from this, that b is the support for a in the domain of u. In the paper [6] there are several other examples of how inference can be used if certain properties hold for the constraints.

To maintain AC during the search all that is done is that one of the AC algorithms is applied to all unassigned variables at every assignment step. Therefore, at each step of the maintaining arc consistency algorithm we need to do three updates. A step consists of a choice of variable x and then an attempt to find a value for it. We pick a value v and first we need to check that no non-binary constraints[1] are violated by the combination of the label `<x,v>` with the existing assignments. Then we need to do the FC stage, by removing all values that are inconsistent with the current label from the domains of the unassigned variables. Finally, the remaining problem (all the unassigned variables) is made arc-consistent by one of the AC algorithms described above. If the first check does not fail or the second two processes do not make any domain empty then that step is completed. However, if this is not the case new values are tried until it is the case or D_x becomes empty and backtracking to the previous step must occur.

Solver [85] combines all three process by altering the way steps are taken. Each step is set up as a *choice point* which opens two branches. The first branch is to constrain a variable i to have a certain value j (this is an assignment). The effect of this constraint is propagated and if a fail occurs then the second branch is tried where a constraint removes j from i. The AC maintaining process is based on AC-5. At each choice point the entire state of the algorithm is saved with all the domains of the variables. If the algorithm backtracks to the choice point the domains are reinstated as they were.

Since Solver is based on AC-5 the way that constraints perform propagation is open

[1] These are the non-binary constraints that are not used in the AC algorithm

and this allows users to develop their own constraints as well as providing an extensive collection of predefined ones. Solver gives a base class for constraints and the user specifies how it will propagate. In Chapter 5 we will see examples of these.

2.4 Variable Ordering

The order in which variables are assigned values can greatly affect the number of fails an algorithm has before a solution is found. In some problems there may be a natural problem specific order. However, there are several general methods. Some of them are discussed in the following section. These often work on the way variables are constrained and how variables are related to each other by constraints. They are classed into two types: static orderings that are decided at the start of the search and do not change and dynamic orderings which may change during the search. Dynamic orderings rely on extra information being generated during the search and so require the domains of unassigned variables to be altered due to the search. For example, if arc-consistency is maintained.

2.4.1 Fail first principle or the smallest domain first ordering

Arguably the most popular example of dynamic variable ordering was introduced by Haralick and Elliot in [51]. The idea was to assign values to the variables that are most likely to cause failure as early as possible rather than later in the search. This would with the aid of constraint propagation in theory cut off fruitless branches early, thus saving search steps. This is called the fail first principle. The way this was implemented was at every step to choose the variable with the smallest domain. The domain size was taken as an indication of how hard it would be to find a value for the variable. This ordering is successful on many problems. However, work by Smith and Grant [98] to use a more accurate indication of how hard a variable is to satisfy had worse results. They concluded that it might not be the fail first principle that is behind the success of the smallest domain first ordering. Smith and Grant give a simple possible reason for the success of the ordering,

by putting the smallest domains first the size of the search tree is reduced. However, this cannot explain the aspect of the ordering that Sabin and Freuder discovered [90]. They used a FC algorithm combined with smallest domain ordering. This was tried on several problems with and without making them arc-consistent in a preprocessing stage. The results showed that on several of these problems the preprocessing actually made the algorithm perform much worse. They concluded this was due to the ordering as the behaviour did not exist when the FC algorithm was applied without the ordering. Much of the work on this has been done on problems where the domains at the start are all of the same size. So the lack of performance might be put down to having variables with different domain sizes before search begins. Since many practical problems have variables with different sized domains the effect of this is of notable importance.

Figure 2.3 shows how the search with dynamic variable ordering and MAC differs from the simple form of search shown in 2.2. When variable V_k is assigned a value it is moved to the assigned variables and V_m is chosen by some heuristic to be the next *current* variable. After each step, variables that have their domains reduced to one value are bound, i.e. assigned that value. An example of this is shown in the figure, when variable V_i was assigned a value, constraint propagation set the value for V_j.

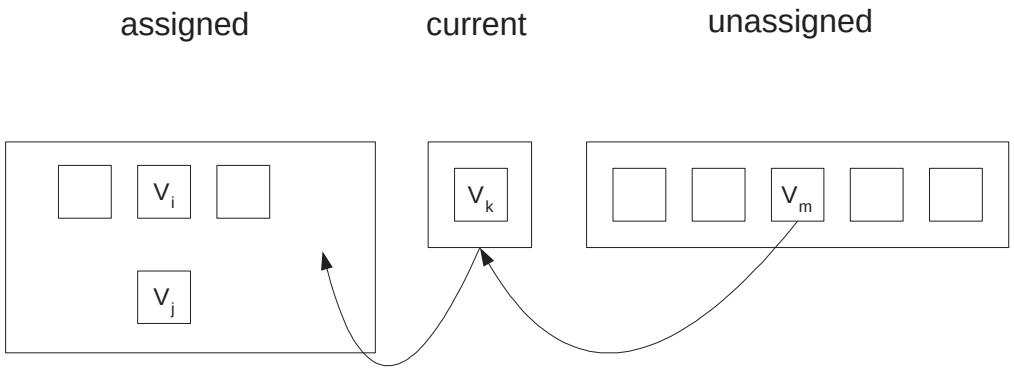

Figure 2.3: Search with dynamic variable ordering and MAC

2.5 Value ordering

Value ordering is useful when we are interested in finding a single solution. If we were after all solutions value ordering would make no difference in chronological backtracking. The way the variables are ordered and the amount of constraint propagation affects the choices of values. If there is a large amount of constraint propagation done after each value assignment then failures can be found quickly and so value ordering is less important. However, if the constraint propagation is not adequate wrong choices of values can lead to a great deal of fruitless search and backtracking. So it can be important to consider a value ordering heuristic.

As stated in [106] the idea is to pick the value most likely to be successful, to reduce backtracking. One way to assess the chance of success is to pick the value which conflicts with the least number of values in the domains of unlabelled variables. There are several variations on this theme. The method of Geelen [41] and the method of Keng and Yun [63] both temporarily assign all the values in turn for a variable and apply forward checking. Keng and Yun then choose the value according to the number of values that would be removed by FC. It uses the percentage loss of values from the domains of unassigned variables. This is similar to Geelen's method which uses the domain sizes of unassigned variables after FC reduction. The real difference in the methods is how they combine these cost elements that come from each of the unassigned variables. Geelen uses the product of them and Keng and Yun uses the sum. In the Keng and Yun method all assignments that would overall remove the same number of values have the same desirability. For example, removing 3 values from one domain and 2 from another is the same as removing 0 values from one domain and 5 from another. However, Geelen argues this should not be true because a problem that has mostly large domains with a few very small domains will be harder to find a solution for than a problem which has all average sized domains. By using the product of domain sizes the two different removals in above example will lead to different evaluations.

A further method is described by Minton [78]. This uses a full assignment of variables

where constraints may be broken. This is used to rate values in the current variable to assign a value to. The less conflicts the value with assignments in the full assignment the higher the rating. At each step of the search the full assignment is reduced to the variables that have not already be assigned a value. This uses the min-conflict heuristic which is described in Section 3.3.

There is a condition where the succeed-first strategy will not be useful. This is when all the values must be chosen at some point and the only choice is which variable is assigned to which value. Smith [96] shows an example where this is the case and suggests applying the fail-first principle, choosing the values that are most constrained first.

However, even more so than variable ordering, problem specific orderings are often the best. This is because general purpose value orderings described above are expensive time wise, as they require extensive consistency checks. We will see below in the next section how greedy heuristics can be used for value ordering.

2.6 Optimisation

When all solutions are not equal and some are desired more than others, often the best (optimal) or as close to the best solution as possible is desired. In these types of problems a solution may have an associated "cost" that we are trying to minimise or 'profit' we are trying to maximise. There will be an objective function which maps every solution tuple to a cost. If we are requiring a profit we can use the negation of the objective function to provide a cost to minimise. A naive approach would be to find all the solutions and then choose the best from them. However, the amount of searching can often be reduced. When a solution is found the cost of the solution is stored as a new bound on the optimal cost. When building the next solution a partial cost can usually be maintained. If this breaks the stored bound then the current partial solution cannot produce a better full solution and backtracking occurs. The stored cost bounds the cost of future solutions. This process is called branch and bound. Even with this reduction the problem may have

to be solved several times and on hard problems this can be very time consuming. The closer to the cost of the optimal the original bound is the less searching has to be done. So using heuristic orderings is a good idea to get as close as possible to the optimal cost at the start.

2.7 Modelling

Modelling a problem as a constraint satisfaction problem is probably the hardest part of the research area to produce general methods for. This is the consensus of many people active in the area of constraint satisfaction and is highlighted by Freuder [39]. Sabin and Freuder have worked on automating the modelling process [91] but the work is far from being practically usable. The hardness of the task is partly due to the flexibility in how a problem can be modelled and that each problem once modelled can be reformulated and extended in numerous ways. In this section we will look at reasons why certain representations can be better than others. The basic model must have one feature, every solution to the CSP must give a solution to the real problem[2]. However, further questions need to be asked of the model. Here are several of these:

1. What is the size of the CSP?

   The size of the CSP can be measured by the number of combinations of possible assignments. So this is the product of the sizes of domains of all the decision variables. There may be non-decision variables in the model where the actual value of them does not relate to the actual problem. These are normally used in conjunction with constraints to constrain decision variables. None of the algorithms discussed in this chapter would search all possible values for decision variables and the forward looking ones would prune some branches of the search tree through constraint propagation. However, the number of possible assignments is still a measure of how hard the problem will be to solve as long as it is taken in conjunction with the other

[2] Although a solution to a CSP could be a solution to a sub problem of the real problem or there could be some repair techniques

measures. So it is logical that choosing a representation on its smallness is a good judge of how good a representation is.

2. How easy are the constraints to implement efficiently?

   It is easier to propagate reduction done by binary constraints than by higher arity constraints. It is very expensive to make a non-binary constraint arc-consistent (in general) and the more variables involved the more expensive it is. There are some constraints for which specified algorithms exist, for instance the all-different constraint but these algorithms are still expensive. So a model that has only binary constraints is more favourable than one that has ternary or higher. Even though in theory higher order constraints can be converted to binary constraints in practice this often will not result in a good model. However, it may be possible to find a model which has smaller arity constraints than the original. A further consideration is the number of constraints and the amount of memory each constraint requires.

3. How close are the variables to the real objects they are modelling?

   This is a little harder to define than the previous two aspects as it is not quantifiable. The more the variables and values can be directly associated with the physical objects in the problem the easier it is to create problem specific heuristics. It will also allow any problem structure to be seen more readily and possibly allow the problem to be reformulated to improve the model. Another benefit is that it makes it easier to explain to non-computer scientists. This is particularly useful if working with the people who used to solve by hand the problem that we are modelling. This will not only allow better feedback but also a greater chance of acceptance of the system. For example, manual schedulers are far more likely to be happy with a scheduling tool when they know the basics of how it works.

4. How easy is it to apply general heuristics to the model?

   Certain ways of representing a problem as a CSP allow the direct use of some of the general heuristics described in this Chapter. Others will need to adapt them to fit the model.

Often a solution is a pairing of objects in the real problem. For example, in the n-queens problem there is a pairing of queens and squares. In these cases it is possible to have either of these objects as the variables. The n-queens problem can be formulated with the variables as the queens and the squares as the values. It can also be formulated with the squares as the variables with a binary domain of 1 to indicate a queen is present or 0 to show one is not. The size of this representation is $(n \times n)^2$. How the constraints are represented are different in each model. However, in the second method further constraints need to be added to ensure only n queens are placed on the board.

Dincbas *et al* [29] model a problem where the objects can be directly swapped so the variables and values can be interchanged. There are 4 of one object and 72 of the other. So the size of the problem could be 4^{72} or 72^4. So by formulating the problem where the size is 72^4 a reduction in the size of the problem is achieved. So this type of remodelling can affect how the constraints are represented and how many constraints there are (item 2) and the size of the problem (item 1). The third case described in Section 5.3 will show that as well as the previous two aspects the remodelling can affect how general heuristics can be applied (item 4).

It is worth noting that the n-queens problem can be represented better by taking advantage of the structure of the problem. We can see that every row must have a queen on it and so we can have the row as the variable. The domain of a row will be the columns. The size of the problem is smaller than having the queens as variables, n^n instead of $(n \times n)^n$. It does have a larger size than using the squares as variables, $(n \times n)^2$ but it removes the need for several constraints e.g. the constraints added to stop more than one queen being on each row. This comes from the fact that the variables and values are directly related to the physical objects of the problem (item 3).

2.7.1 Symmetry

Another important consideration in modelling is symmetry. This is where several solutions to a CSP represent the same solution to the actual problem. This leads to problems sizes

being much bigger than they needed to be because certain combinations are the same and need not be tried more than once (item 1). Work was done by Puget [84] to add constraints to eliminate symmetry. A common example of symmetry can be seen in the n-queens problem. Since a chessboard is square if the top of the board is rotated, the side previously to its left becomes the new top. So solutions that can be mapped to each other by rotation or reflection are the same solution. A way of solving this problem that is applicable to many other instances is to artificially discriminate the variables. Add a constraint that specifies first queen must be closer to the top left corner than the second queen.

2.7.2 Adding extra constraints

There are other cases as well as symmetry where adding constraints can improve the search. This is done by adding what are called redundant or implied constraints. ILOG Solver's manual [58] defines these as constraints that make explicit a logical consequence of other constraints of a problem. An example of an implicated constraint can be shown in the graph colouring problem. Figure 2.4 shows that variables A and C must have the same value so a constraint can be introduced to inform the search of this before it starts.

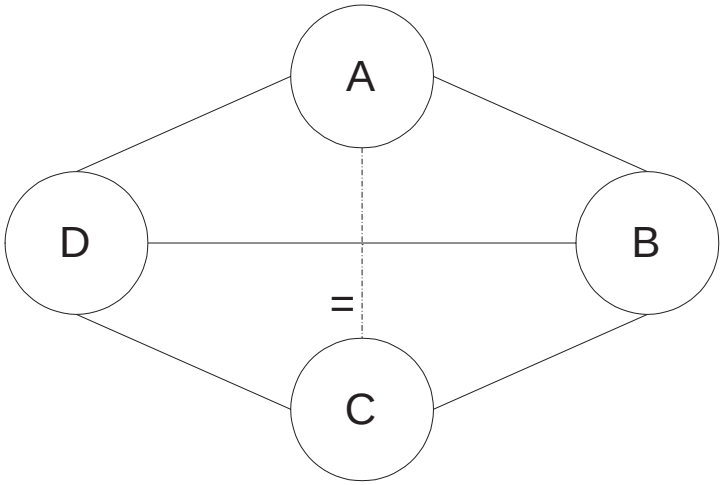

Figure 2.4: This shows a constraint graph of a graph colouring problem. The dotted line shows an implied equality constraint that variables A and C must be equal. All the other constraints are inequality constraints.

This is similar to some of the implied constraints that Sqalli and Freuder uses in [101]. Freuder also suggests the use of implied constraints to replace higher order constraints to improve constraint propagation [39] (item 2). At the start of this section on modelling it was stated that the first thing needed in a model is that all the solutions to the CSP are solutions in the real problem. However, does the reverse have to hold? If we are after only a single solution it may be advantageous to remove some of the solution as long as we also reduce the size of the problem. In large problems time limits may in practise remove many possibilities as there may be no time to explore all avenues. However, we must ensure at least one solution remains. So we can add extra constraints to cut further the search space even if they may cut out possible solutions. This is further investigated in Section 5.6.2.

2.8 ILP vs. CP and evaluating algorithms in general

There has been many studies comparing ILP and CP [86, 22, 97, 87, 83]. Many have proposed ways of combining ILP and CP to take advantage of both techniques [8, 56, 87, 31, 33, 32].

From these studies several aspects of each technique have been highlighted. The first aspect that is easy to see is that in ILP constraints must be linear whereas CP constants have a much larger range of expression. CP seems to do better on problems that can take advantage of the efficient general constraints that have been implemented, foremost the all-different constraint (constrain a set of variables to have different values) and to a lesser degree constraints to remove symmetry. The all-different constraint is efficiently implemented in CP but in ILP applying constraints to do the same job vastly increases the model size. This is shown in [97] and later on a similar problem in [22]. Adding constraints to remove symmetry in CP reduces the search space and removes unnecessary searching. However, adding similar constraints to a ILP model will not cut the search space but increase the model size, this is seen in [22]. This illustrates one of the main differences between the two methods. ILP globally cuts the search space whereas CP locally reduces the search space. Therefore where the search space can be easily cut globally by good lower

and upper bounds on optimisation problems then ILP usually performs well. However, if this is not possible, as in the job shop scheduling problem[3] [11], ILP may find it hard to solve problems. CP depends on the constraints of the model providing enough propagation to reduce the search space.

In evaluating the effectiveness of ILP and CP on practical problems, we wish to put forward several warnings. Moreover some of these apply to evaluating algorithms in general.

1. Practical is not always practical.

   Often so called practical problems are only approximations of real world problems. Sometimes side issues are ignore to make the problem easier for the community to grasp. Beck *et al* [3] for example warn about the obsession with only optimising make-span time in job shop scheduling. They cite several other restrictions that may need to be considered in a real scheduling situation. This over simplification of real world problems may make CP seem worse than ILP in general. This is because CP has a more flexible language for defining problems than ILP and so side issues are more likely to cause problems for an ILP approach than a CP approach. This issue may be compounded by the fact that if the problem was formulated first by a researcher in a particular field they may introduce bias. The paper on CSPlib [45] discusses how bias may be introduced and therefore specifies that real world problems should be specified in a natural language so as to limit any bias in formulation.

2. Number of problems tested

   It is often hard to find enough suitable instances of an industrial problem. Whereas random problems can be generated in their hundreds, many of the practical problems have few instances. For example, Darby-Dowman and Little [22] show results on crew scheduling but only have 5 instances of the problem. There is however, little that can be done about this except keeping it in mind when viewing results.

3. The amounts of effort or expertise for each technique

[3] The job shop scheduling problem is an industrial problem involving assigning a number tasks to machines on a factory floor.

> In some of the comparisons very little effort is put into the CP and ILP algorithms to solve the test problems. For example, Rodosek use no variable or value ordering in their CP representations. Similarly with the ILP approach simple CPLEX standard algorithms are used. Often the difference between ILP and CP is so overwhelming it is unlikely that there will be a change if time is taken to improve each algorithm but this field should display the same rigours of science as any area of physics or chemistry.

Further to these Hooker [55] puts forward an argument that completive experiments in general are difficult to judge fairly and moreover may not be productive, as they do not give the reasons why certain algorithms are faster or slower than others.

2.9 Local search

In this Chapter we have discussed aspects of systematic search on constraint satisfaction problems. There has been some research on how aspects of systematic search can be related to local search techniques. Several papers have been published on adding consistency to local search techniques [62, 102]. Another interesting aspect, symmetry's effect on local search is discussed in Section 6.1.

2.10 Summary

There are many other basic search methods and hybrids of the above methods. There are also numerous heuristics and variable and value guides. Those that have been given here have been selected to relate to the research in this thesis. A fuller account of the range of work on constraint satisfaction is given in [106].

Modelling problems as constraint satisfaction problems in an efficient way often needs informal heuristics and creative input by an expert in the field. There are general guides

but even these are open to debate.

Evaluation of models, algorithms and techniques as a whole (e.g. CP vs. ILP) is not always straight forward as often empirical evidence is need to be used. Such evidence by its nature is open to error and interpretation.

The rest of the thesis will examine how the explained research in this chapter and the methodology issues discussed can be extended and developed to produce driver schedules.

Chapter 3

Local Search for Constraint Satisfaction Problems

3.1 Introduction

There are numerous local search methods for solving constraint satisfaction problems. An overview of several is given in [49]. Presented here are some of the more popular methods and their origins. The main focus of this chapter is the developments that lead to the creation of the local search method GENET. GENET is the local search algorithm used in Chapter 6 to construct driver schedules.

Informally, the basics of local search consists of first creating a possibly flawed solution to a problem. This can be done either by random assignments or by heuristics. Then the solution is iteratively altered in small ways to improve the solution. These are called local moves as they consider only a small part of the solution and improve that part. There may

be several possible moves and these will be assessed on a measure of improvement that may be different for each problem, for example in a CSP the measure of improvement may be the increase in the number of constraints that are satisfied after the move is made. There is normally some randomness incorporated into the choice of what local move to make at each iteration. This protects the solver from following a set path that may never lead to a feasible solution. If the solver is run several times it may produce different solutions. One important aspect to note, is that the local search technique will always produce some sort of solution even if it does not find a feasible solution. This follows because at every stage of the search a solution exists.

One difference between the local search approaches and the systematic approaches reviewed in Chapter 2 is that given time the systematic approaches will always find a feasible solution if one exists. On the other hand, due to the stochastic nature of local search it may never end up finding a feasible solution but keep cycling through infeasible solutions. However, in practice large problems and time restrictions may negate the ability of a systematic complete search to always find a feasible solution. If no feasible solution is found then the complete search will produce no solution at all. In these cases local search techniques are often used to find as good a solution as possible. Furthermore, local search techniques used for optimisation cannot prove that they have found an optimal solution, unlike complete systematic approaches. Therefore, the stopping criterion for a local search system may be a limit on number of iterations or a time limit. Once stopped, the best solution produced is given as the final output.

3.2 Neural networks

Artificial neural networks have attracted much research because they are based on the human brain. This provides advantages such as learning and as we will see below some parallel processing can be done to speed up the algorithm. There are many good books describing the general field of neural networks, one of which is [1].

Neural networks consist of a large number of *neurons* or *nodes* which communicate via *weighted* connections. The neurons send inhibitory (negative) or excitatory (positive) signals via the connections. These signals range from -1 to 1 in the analog version of the system but we will restrict ourselves to describing the discrete version where the node can either be *on* or *off* sending a signal of 1 or -1 respectively.

A Hopfield network is a neural network where every node is connected to every other node but not itself. A diagram of a Hopfield network is given in Figure 3.1.

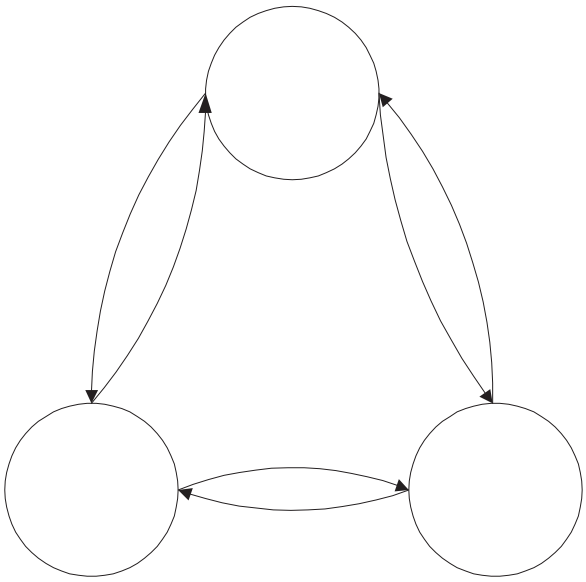

Figure 3.1: 3 node Hopfield neural network

The connections are weighted and this weight is symmetrical, i.e. the weight w_{ij} of the connection from node i to node j is the same as w_{ji}, the weight of the connection from j to i. The output of a node is given as the input to all the other nodes multiplied by the weight associated with each connection.

Every state of the network an associated energy value E. The energy function is defined as: (notation from [1])

$$E = -\frac{1}{2}\sum_{i}\sum_{j\neq i} w_{ij}x_i x_j + \sum_{i} x_i T_i$$

where x_i is the state of the node (ranging from -1 to 1). T_i is the threshold of a node. In a hardware implementation this is an external input supplied to each node.

So there is a energy level for every state the network can be in. This creates an energy landscape. An energy landscape is shown but only in one-dimension in Figure 3.2. This

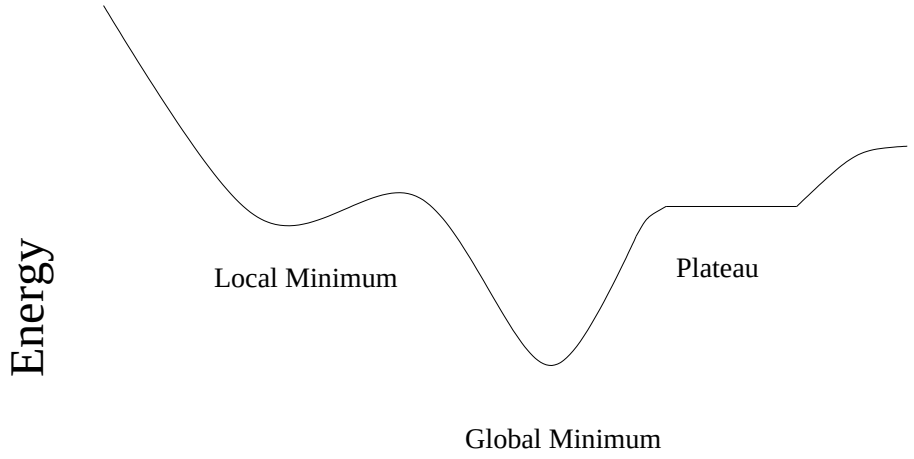

Figure 3.2: Diagram of energy function

landscape representation can be produced for all local search methods. There may be several global minima as several states may have the same energy level. When states of the same energy level are adjacent to search other we call them a plateau.

The network can be updated in one of two ways. Either all the nodes are updated in parallel or they are updated sequentially, a node is picked at random and then updated. The main difference is that in the sequential case the effects of the update of one node can influence the state of the next node that is updated whereas in the parallel version all the nodes update independently. Each update of a node consists of turning the node on (1) if the input is above the threshold and off (-1) if below. When we use the Hopfield network to solve CSPs the threshold is set to zero and so if the input is above this it will be set to on.

Tagliarini and Page [103, 104] used a Hopfield network to solve a CSP, specifically the

n-queens problem The neurons represent the squares on the chess board. If there is a constraint between the squares there would be an inhibitory weight. There is also a component of the weights to guide the network towards a state where there are exactly n queens on the board.

A major flaw in this approach to solving CSPs was that the network would become "stuck" in local minima. This would mean that constraints would be broken and so the solution might not be useful to the user. Moreover, there may be states where variables might not have a value assigned to them. The common way of dealing with this was to restart the network every time it reached a local minimum. However, on hard problems this approach is unlikely to find a global minimum as all the effort put into a previous search is lost when the new search starts. Further work, by Adorf and Johnston [61] solved at least part of this problem. Their guarded discrete stochastic (GDS) network ensured that a variable would always have an associated value in the network.

3.3 Min-conflict heuristic

In 1992 Minton *et al* [78] investigated why the neural network approach (specifically the GDS network) was doing better on certain problems (e.g. the n-queens problem) than the backtracking algorithms of that time.

The first argument considers the non-systematic nature of the GDS approach and the structure of the search space. If the search space has solutions clustered together rather than spread evenly, a systematic search may take longer than a non-systematic search to find a solution. This is explored in their paper by using a purely random search, the Las Vegas algorithm, which they show performs better than a simple backtracking search on the n-queens problem. However, the GDS network outperforms the Las Vegas algorithm so there must be further explanation for the success.

The second argument is that having a whole assignment to a problem gives knowledge

that is not available to a constructive backtracking approach. So out of the GDS network a simple heuristic was distilled to demonstrate the reason for the success of the network, the min-conflicts heuristic:

Given: A set of variables, a set of binary constraints, and an assignment of a value for each variable. Two variables *conflict* if their values violate a constraint.

Procedure: Select a variable that is in conflict, and assign it a value that minimises the number of conflicts. (Break ties randomly.)

Empirical evidence obtained from [78] using the min-conflict heuristic for hill climbing [1] showed that the heuristic obtained similar results to the neural network, so supporting the argument that the network's success is due to the principle captured by the min-conflicts heuristic.

Using the min-conflicts heuristic instead of the GDS network allows more flexibility in the way the search is conducted. For example in [78] a backtracking system is implemented using the min-conflict heuristic for variable and value ordering.

The local search min-conflicts heuristic worked well on problems such as the n-queens problem, graph colouring problems and the real world problem of scheduling the Hubble Space Telescope [78]. However, still present was the problem of getting stuck in local minima. In section 3.5 there is discussion on methods for escaping local minima but first we will introduce another algorithm used for solving CSPs.

3.4 GSAT

GSAT [95] is a greedy local search for solving propositional satisfiability or SAT problems. To explain this the following is defined:

[1] Hill climbing is used to find a maximum in the search space and gradient descent is used to find a minimum. However, maximising the negation of the objective is the same as minimising the objective function so these terms will be used interchangeably

1. A literal is a propositional variable or its negation. E.g. A or $\neg A$
2. A clause is a disjunction of literals. E.g. $(\neg A \vee B \vee F)$
3. A formula in conjunctive normal form (CNF) is a conjunction of disjunctions. E.g. $(\neg A \vee B \vee F) \wedge (B \vee \neg C \vee \neg D) \wedge \ldots$

A SAT problem is: given a CNF formula find an assignment of true or false for its variables (a truth assignment) that satisfies the formula. CSPs can be represented as SAT problems[2] and so GSAT can solve them. The search method starts with a random truth assignment. Then iteratively: change ("flip") the variable's truth value that leads to the largest increase in the total number of satisfied clauses. After a user defined number of flips (MAX-FLIPS) the search starts over with a new random assignment. This whole process is repeated a given number of times (MAX-TRIES). The full procedure is given in Figure 3.3.

GSAT
where α is a set of clauses

For MAX-TRIES
    $T :=$ a random truth assignment
    For MAX-FLIPS
        if T satisfies α **then return** T (solution)
        $p :=$ a propositional variable such that a change in its truth assignment gives the largest increase in the total number of clauses of α that satisfied by T. Breaking ties randomly.
        $T := T$ with the truth assignment of p reversed.
    end
end
return "no satisfying assignment found"

Figure 3.3: Pseudo code for basic GSAT procedure

Both min-conflicts and GSAT allow sideways moves, the current solution is allowed to move to another solution with the same energy level. This lets the procedure traverse plateaus in the energy landscape, see Figure 3.2. By doing this the search can find ways

[2]CSPs represented as SAT problems can have inflated search spaces, see section 3.6.4

off the plateau and continue gradient descent. GSAT actually allows uphill moves, if there is no move that increases or retains the number of clauses satisfied. However, this is not enough to escape a local minimum. To do this the heuristic has not only to move out of it but try not to "fall" back into it.

3.5 Methods for escaping local minima

There are several approaches for escaping local minima in heuristic improvement methods. These same methods can often diversify the search. These can be categorised into two types of approach (or a mixture of the two): those that add randomness such as Simulated annealing [64] and those that restructure the neighbourhood such as Tabu search [46] and weighting approaches [80, 93].

3.5.1 Simulated annealing

Simulated annealing has been used for solving CSPs [73]. The standard simulated annealing process works as a gradient descent neighbourhood search allowing uphill moves with a certain (possible varying) probability. A move consists of choosing a neighbouring state at random and if this state has a lower energy then choose it. Otherwise choose it with a probability $P = e^{-\Delta E/T}$, where E is the energy and ΔE is the change in energy that would be produced by the move. T is a temperature level, which may be constant or decreasing during the search. This value affects how likely a non-improving move is made, the higher T the more chance.

3.5.2 Tabu Search

Tabu search like GSAT allows uphill moves if no improving move can be made, yet it explicitly tries not to "fall" back into local minima. It does this by making previous states (and related states) Tabu. It stores a list of these Tabu states and dynamically updates

this list as the search proceeds. This is a flexible meta-heuristic and can be implemented in many ways and hybridised with many other search methods. An overview of these can be found in [46]. The basic model is applied as follows. Start with an initial solution (possibly randomly generated). Move to the best available state even if this is a non-improving move. Update the Tabu list. In the basic model this can be done by adding the previous state to the Tabu list and removing states after a specified number of moves. Repeat this until a set number of steps is reached or no moves are available.

3.5.3 Escaping local minima in GSAT

Simulated annealing and similar approaches have been incorporated into GSAT [92, 94], one such approach was GSAT with Random walk. The principle is outlined as:

With probability p, pick a variable occurring in some unsatisfied clause and flip its truth assignment

With probability $1 - p$, follow GSAT, i.e. pick randomly from the list of variables that gives the largest decrease in the total number of unsatisfied clauses.

A further method introduced in [92] did not directly escape local minima but altered the search space to remove them. It was discovered that, in some problem instances, after each run the same set of clauses would remain unsatisfied (an example of this is given in [92]). To combat this a weighting system to increase the importance of certain clauses was introduced. At the end of each inner cycle of GSAT (see Figure 3.3) the cost of violating a clause that is violated in the current assignment is increased. This work was later built on in [14] where a similar effect was produced by adding extra clauses instead of changing weights. It is claimed that this new method works better than the previous method. This claim is founded on empirical evidence and in the paper a possible explanation is given.

The best version of GSAT out of the ones shown was according to Selman *et al* [94] GSAT with Random Walk. However, this is debatable as in [13] it is concluded that

GSAT-weighting is the best method. The reason for the debate of which method is best is because performance is based on empirical testing on problem instances. For different classes of problems different solvers may be better. There have been several explorations of various versions of SAT solvers(e.g. [44]).

3.5.4 Breakout Method

A similar approach to the weighting approach of the last section described above was described in Morris [80].

In the min-conflict heuristic, the cost or energy function is the number of constraints violated. In this method each constraint (represented as a nogood constraint) has a weight, initially 1. The cost function is the sum of all of the weights of the violated constraints. A standard gradient descent search is used until a local minimum is reached. Then the weights of the current violated constraints are incremented until the current state is no longer a local minimum. The search then continues. This method differs from the GSAT-weighting in that as soon as a local minimum is found the weights are increased rather than after a fixed number of iterations.

Morris proves that if this increase of weights only affected the current local minimum then the algorithm would be complete and so given enough time would always find a global minimum. However, the weighting effect deforms other parts of the space and this makes the search incomplete.

3.6 Description of GENET

GENET is a Neural Network adapted from a Hopfield Network described above. The network can represent a constraint satisfaction problem. It could be implemented into hardware and the design for this is detailed in [122]. However, it has been successfully used as software simulation and this is what is described here.

Each neuron (or node) represents one label. The label nodes corresponding to a particular variable form a cluster. Each node can be in an on or off state. If the node is on, then the associated variable and value are assigned. Therefore, only one node in each cluster is allowed to be on at any time, as a variable can only have one value. The node's state is governed by the input to the nodes in its cluster. In turn the node has an output and this is 1 if the state is on and 0 if not.

Binary constraints are represented by connections showing a nogood association between label nodes. These work in a similar way to the connections in a Hopfield network. Consider two labels whose representing nodes are X and Y and which are prohibited from being on at once by a constraint. The connection denoting the constraint has an associated inhibitory (negative) weight. This symmetrical connection takes the output of node X (Y), multiplies it by the associated weight and adds it to the input of Y (X), where w_k is the weight and starts at -1. A diagram showing an example of GENET is given below in Figure 3.4. Here variables A, B and C have domains of {1, 5, 7}, {8, 14} and {5, 9, 11} respectively. There is a connection between the nodes denoting `<A,1>` and `<B,14>` (further referred to as A_1 and B_{14}) and this represents a binary constraint restricting the assignment of A to 1 and B to 14. Other binary constraints are similarly represented. So if the nodes A_1, B_{14}, C_9 and were on, the input would be: -2 to node B_{14}, -1 to nodes A_1, B_7, C_9 and 0 to the rest.

The search method is based on the min-conflicts heuristic described above. It starts with a random assignment of values to variables. In the network a random node in each cluster is set to an on state. Then all weights are initialised to -1. For each iteration of GENET the variables are cycled through in a random order. For each variable cluster the label node with the highest (closest to zero) input is turned on. Ties are broken as follows: If one of the nodes with the minimum input was previously on it stays on, otherwise ties are broken randomly. This process is repeated until one of three situations occurs:

1. All the labels that are on have an input of zero (a global minimum has been found).
2. No improving move can be made for any of the variables (a local minimum has been

reached). Dealing with this will be described in the next section.

3. Some predefined limit on the number of iterations or the maximum time has been reached.

3.6.1 Escaping local minima

When caught in a local minimum GENET increases the importance of the constraints that are violated in that assignment i.e. it decreases the weight of the constraints involved by 1. So the energy landscape is altered and the local minimum is raised or "filled in" and descent can continue. This process is called "learning" because by performing this operation GENET will discover which are the hard constraints to satisfy, giving them more importance. Learning also leads to the heuristic exploring a wider search space than it would otherwise, because features of previous assignments in local minima are penalised and so are less likely to recur. This is similar in effect to the Tabu [46] process.

So the final basic GENET algorithm is in Figure 3.5.

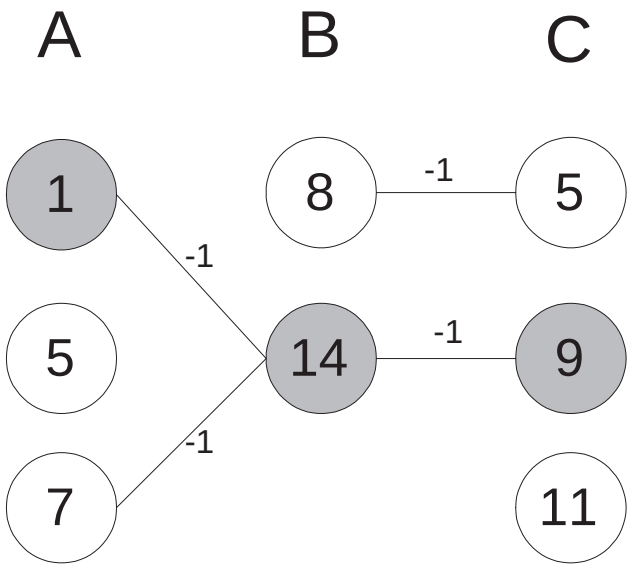

Figure 3.4: Three variable GENET network

3.6.2 General considerations

A consideration is whether to allow non-improving (sideways) moves i.e. changes that do not reduce the number of violated constraints. The basic model described above does not allow sideways moves: a node that was previously on which has the minimum input in the cluster stays on, even if other nodes have an equal input. The advantage of not allowing sideways moves is that this guarantees convergence. Given enough time the system will always find a local or global minimum, whereas if sideways moves are allowed the network may never stop changing states. Davenport [23] notes that GSAT successfully uses extensive sideways moves. A problem with sideways moves in GENET is that when we make a move we are only considering one variable. It may be that there are improving moves that can be made with other variables and by making a sideways move we may remove this possible improvement. Although a similar state may occur without sideways moves being used, there may be a better move missed. Davenport developed three strategies for allowing sideways moves: None (only learning), limited and full. The full sideways moves version allows all node clusters to change the node with the on state, even if there is no conflict (i.e. the node that is on has zero input). If the network stays in the same state after two consecutive cycles learning is invoked. The limited approach allows the

GENET

Randomly turn on one node in each cluster

Repeat :

  Repeat : Randomly order the clusters

    For each cluster in order

      Out of the set of nodes with highest input in the cluster; retain previously on node if member, else turn on a random member.

  until convergence (no label nodes changed state in a cycle)

  if in a local minimum (not all inputs to on label nodes are zero)

    Learn

until in a global minimum or resource limit reached

Figure 3.5: Pseudo code for basic GENET model

same moves as the full approach. However, it only allows two consecutive cycles without changing the overall energy before learning occurs. This method has both the advantage of guaranteed convergence and the advantage of sideways moves. Davenport experimented with several classes of problems: the n-queens, random binary, graph colouring and planning. From these Davenport concluded that no one system is better than another. For example, allowing full sideways moves is best for the n-queens problem while for planning problems allowing no sideways moves is best. A possible reason for planning problems benefiting from not using sideways moves is that they are highly structured and whether a label causes conflicts or not is strongly based on the choice of other labels.

3.6.3 Non-binary constraints

All non-binary constraints can be represented as binary constraints [106]. However, this tends to hugely inflate the size of the problem. So there is a need to express more general constraints in GENET. For non-binary constraints the architecture of the model has to be changed. *Constraint neurons* are added which represent the non-binary constraints. Davenport [25] introduced ways of dealing with several general non-binary constraints. A basis for these non-binary constraints and some specific constraints will now be described.

The input to a constraint node is the unweighted sum of the outputs of all labels that violate the constraint. The output is weighted just like the binary constraints. A weight is stored for each constraint node. So the constraint node - label node connection is non-symmetrical, unlike the label node - label node constraint connections. The input to a constraint node directly affects its state (S) and has to be set up so that it acts in the following way. If the constraint is being broken, S will be positive. If it may be broken by one variable changing value S will be zero. Otherwise, S should be negative.

Figure 3.6 shows a model of a possible non-binary constraint. The constraint could penalise node $< A, 1 >$, $< B, 8 >$, $< C, 9 >$ and/or $< C, 11 >$ as it has connections to these. There is one weight -1 stored in the constraint.

The learning mechanism updates the weight in the same way as with binary constraints. The weight of the constraint is decreased by 1 if it is in conflict at a local minimum. This weight is associated with all the all label nodes connected to the constraint node and so affects the input of all of them.

Davenport *et al* illustrate some specific constraints in [25] and more in [23]. Here is a summary of two of these:

1. The Illegal (or nogood) constraint restricts the use of particular compound labels. The constraint is given a k-compound label L that is invalid or illegal in a solution. The constraint node is connected to the k label nodes in L. The state of the illegal constraint node S_{ill} is negative if the input I_{ill} is less than $k-1$. This is because even if one label changes state no violation can occur if fewer than $k-1$ nodes are on. However, if exactly $k-1$ nodes are on S_{ill} will be 0, because if the remaining label node in the off state is switched on the constraint will become violated. This is expressed by the equation:

$$S_{ill} = I_{ill} - (k-1)$$

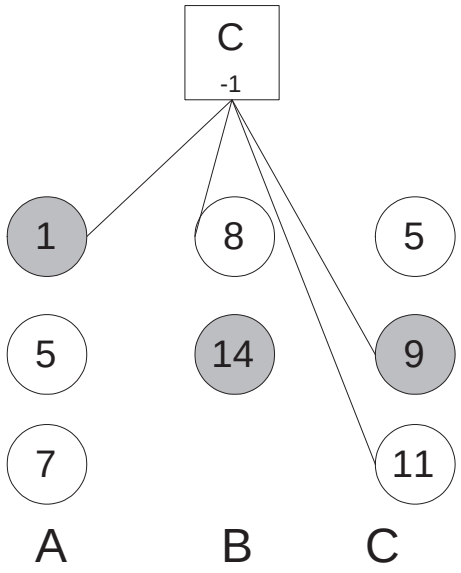

Figure 3.6: The framework of a non-binary constraint in GENET

If $S_{ill} = 0$, i.e. only one node is in the off state, we will discourage this node from turning on by the constraint applying a weighted output to it. The other labels are not penalised, because on their own, they will not cause a violation.

The last situation for S_{ill} is if it is positive, i.e. all k label nodes are on. In this case all the nodes are penalised to persuade them to change state.

The output ($V_{ill} < i, j >$) of the illegal constraint node to each label node $< i, j >$ can be represented by the equation:

$$V_{ill<i,j>} = \begin{cases} 0 & \text{if } S_{ill} < 0 \\ 1 + S_{ill} - V_{<i,j>} & \text{otherwise} \end{cases}$$

where $V_{<i,j>}$ is the state of the label node $< i, j >$.

The Illegal constraint is useful as it can be used to represent more general constraints. As any constraint can be a represent by binary constraints any constraint can be represented as Illegal constraints. The Illegal constraint representation will be of equal or smaller size to the corresponding binary representation.

2. The Atmost constraint is a common constraint and so has been included in CHIP. Given a set of variables Var, a set of values Val and a number N, let L be the set of labels that can be generated from Var and Val. That is, $L = \{< i, j > | i \in Var, j \in Val, j \in D_i\}$. Then the Atmost constraint states that any compound label in the solution must contain at most N labels in L. So only N variables in Var can have values from Val.

   In GENET the Atmost constraint node is connected to all the labels in L. The state S_{atm} is determined as follows:

   $$S_{atm} = I_{atm} - N \text{ where } I_{atm} \text{ is the constraint's input.}$$

   So as in the Illegal constraint if the state is negative no nodes are penalised and if positive all are penalised. However, if the state is zero it is dealt with differently.

When $S_{atm} = 0$ any of the remaining nodes turning on would cause a violation. Unfortunately, if *all* of these remaining nodes were penalised a problem would occur. Unlike the Illegal constraint, in the Atmost constraint a single variable can be associated with several constrained labels. So say a variable i has two values (j and k) in its domain that are in Val. If the constraint state is zero and the label $< i, j >$ is on and $< i, k >$ is off we would penalise $< i, k >$ but not $< i, j >$. So in the next move GENET could switch $< i, j >$ off and $< i, k >$ on. This switch could then happen in reverse in the next move. So the network could oscillate between one node being on and the other on. To remove this problem all label nodes in the same cluster receive the same output from this constraint. If all of them are off then the constraint will output a one multiplied by the constraint weight to all of them to dissuade one of them coming on. otherwise it will output a zero. To summarise, the output for a particular label $V_{atm<i,j>}$ is worked out as follows

$$V_{atm<i,j>} = \begin{cases} 0 & \text{if } S_{atm} < 0 \\ 1 - Max\{V_{<i,k>} | k \in Val\} & \text{if } S_{atm} = 0 \\ 1 & \text{otherwise} \end{cases}$$

In the original work by Davenport et. al. [25] it was stated that for each constraint node there was a separate weight associated with every connection it had with a label node. This idea was dropped in the later work [23] and so this newer version is what has been described above. Only having one weight per constraint node does save memory.

There has also been work by other authors on allowing GENET to handle non-binary constraints. This work saw the emergence of EGENET [72]. This method is similar to the one described above. Some of the differences of note are that multiple penalty values are used for constraints in EGENET rather than a single weight. In a constraint there is a penalty value for every combination (tuples) of assignments of values to variables. Each penalty value starts at -1 for tuples that are prohibited and 0 for others. This allows

greater flexibility in definitions of constraints as the user just needs to define prohibited tuples to generate a constraint. However, this requires a much greater amount of memory then just storing a single weight. So an adaptation was introduced [71] to compensate for this problem. In [70] new constraints were introduced to make EGENET more of a general CSP solver such as CHIP and SOLVER.In the light of the research on EGENET where multiple penalties are used Davenport [23] mentions that only using one weight as opposed to multiple weights can affect the search and suggest it is an area for further investigation. It will be shown later in Section 6.6 that it is not always desirable to have only one weight.

3.6.4 Applications and extensions of GENET

GENET has been successfully applied not only to random CSPs but to several other problems including standard problems such as graph colouring and real world problems such as car sequencing and radio frequency assignment. These use the binary and non-binary versions and several expansions of GENET.

Davenport *et al* [25] claim that GENET is superior to GSAT for problems such as graph colouring. This is shown in experimental results and backed up with the following explanation. In GSAT a problem with N vertices, k colours will require Nk variables to represent it. The domain size of all the variables will be 2. The problem can be represented as a CSP using only N variables with a domain size of k. So in GSAT the number of possible assignments is 2^{Nk} whereas in GENET, it is k^N. So the search space is much larger in GSAT.

The car sequencing problem is a real world problem. Modern cars often have different models with varying features such as sunroofs and air-conditioning. The number of each model required is called the production requirement. On a production line there is a maximum number in any sub-sequence of cars that can be fitted with a particular feature. These make up the capacity constraints. This problem inspired a new neighbourhood strategy for GENET. This was called SWAPGENET [24]. The original representation of

the problem was to have each variable as a position on the conveyor belt. The domain of these variables would be the different models to produce. A normal move in GENET would be to change a position in the conveyor belt to contain a different model. The number of cars of each model to be made are known. So an initial assignment can be created having the right number of models produced even if capacity constraints are broken. The move operator can be changed so that it consists of a variable swapping its value with the value from another variable. This ensures that the production requirements do not need to be implemented as constraints. A further advantage is that it can be proved that the second representation gives a smaller search than the original and so solutions should be found faster. SWAPGENET takes more time for each repair. So although the number of repairs is reduced on easier problems, the time taken to solve them can be greater.

Another real world problem that GENET has been used on is the Radio Link Frequency Assignment problem (RLFAP). Boyce *et al* [10] explore using GENET and Tabu as two techniques for solving the RLFAP. A paper [9] by the same authors with Bouju concentrates on Tabu but gives more detail. This problem will be examined in the next subsection.

Several other authors have extended GENET to deal with standard types of CSPs that the original GENET could not handle. Wong and Leung [124] enhanced GENET to be able to tackle a new class of CSPs; fuzzy CSPs (FCSP). In [17] Cox and Tsang designed a prototype of a GENET that could incorporate continuous domains. EGENET was extended to make use of constraint consistency checking in [102].

3.6.5 Optimisation

GENET was originally designed to find a single solution, stopping once there are no violated constraints. The class of CSPs where solutions can be ordered and the aim is to find the best one are known as constraint satisfaction optimisation problems (CSOPs). A variation of this problem is the Partial CSP (PCSP) where solutions that contain violated constraints are allowed. The constraints that can be broken are called *soft* con-

straints. In this sort of problem, minimisation of constraint violations can be the aim of the search. There may be a hierarchy of constraints and this will affect the preference order of solutions. Since optimisation problems are common, research was carried out to integrate optimisation into GENET. Two general ways of accomplishing this are described in Section 3.6.5.1 and 3.6.5.2

3.6.5.1 The tunnelling algorithm

The tunnelling algorithm was introduced by Voudouris and Tsang in [113]. The idea is to modify the cost function to encode the desired criterion or criteria to optimise. This is done by adding extra terms to the input of each label. In the original model there are only costs for violating constraints and all of these start at the same weighting. In the new version there are additional starting costs for violating constraints and costs for specific assignments. This additional input combined with the original input is called the tunnelling function. So now the cost (c_k^t) of violating a constraint in the tunnelling functions is:

$$c_k^t = \begin{cases} r_k + p_k & \text{if constraint } k \text{ is violated} \\ 0 & \text{else} \end{cases}$$

where p_k starts at 0 and r_k is a fixed cost related to the importance of the constraint. A similar term is added for the labels:

$$l_{ij}^t = \begin{cases} a_{ij} + p_{ij} & \text{if node } < i, j > \text{ is on} \\ 0 & \text{else} \end{cases}$$

where a_{ij} is a fixed cost for each label and p_{ij} starts at 0.

There are two ways in which the tunnelling algorithm can work. The first is called the one stage tunnelling algorithm (1ST). This works just like the original GENET except the

extra terms are included in the cost function. The second, called the two stage tunnelling algorithm (2ST) separates the objective function from the tunnelling function. So the terms of the objective are for the constraint terms:

$$c_k = \begin{cases} r_k & \text{if constraint } k \text{ is violated} \\ 0 & \text{else} \end{cases}$$

Similarly with the label terms:

$$l_{ij} = \begin{cases} a_{ij} & \text{if node } < i, j > \text{ is on} \\ 0 & \text{else} \end{cases}$$

This is done because the tunnelling function can become distorted from the original objective function that is to be minimised. This may cause the algorithm to be unable to find a good solution.

Unlike the original version of GENET the two stage tunnelling algorithm only adjusts the weights of certain terms in the tunnelling function at local minima. This is so that the most important terms are penalised the most and so become less and less likely to be broken. However, to even the process up and so diversify the search, the number of times (the absolute frequency) a term has been previously penalised is considered. So a simple function is instigated to decide which terms to penalise. This is called the Frequency to Cost Ratio (FCR) where:

FCR = Frequency / Cost

At each local minimum a set of terms is constructed consisting of those with the minimum FCR which also contribute to the total cost. Out of this set all the ones with the maximum cost are penalised. There is no indication that experiments were used to derive the relative importance of frequency and cost. They just taking them as equally important in deciding

which terms to penalise.

Another change from the original version of GENET is that the tunnelling version requires extra work to decide how much to penalise each term in order to escape local minima. This is partly due to the new version having two different functions to minimise and partly to deal with the different importance levels (costs) of each term. So the algorithm works out the input to each label for both functions. Then for each variable (v) it finds the minimum label input for both and records the difference (Δg_v). The important criterion for the change in the tunnelling function is that there must be a move (a change of one label to another) available after the weight changes have occurred. However, the bigger the change in the function the further from the original function it becomes. Since the original function is the one that is to be maximised straying too far from it is not desirable. So to balance these two issues the following equation is used for each term:

$$PenaltyAmount = max\{cost, min\{\Delta g_v\}\}$$

By using the cost if it is high enough to make a change possible (i.e. $cost > \{\Delta g_v\}$) the algorithm is more likely to retain the relative costs of the original weighting. This processing takes up a lot more CPU time per cycle than it does when the algorithm just has one function. However, on some of the harder problems the reduction in the number of cycles outweighs this increase in time and the problem is solved in either a quicker time or there is a higher rate of runs that find an optimal solution.

The algorithm was used on: random CSPs and PCSPs, the graph colouring problem, the Radio Link Frequency Assignment problem (described below) and the travelling salesman problem. So it has been shown that it can be applied to a wide variety of problems both random and real world.

It is interesting that on some hard (tightly constrained) non-optimisation CSPs the tunnelling algorithm found solutions on more runs than the original GENET.

An investigation of local search methods for PCSPs and a comparison with a systematic branch and bound method is given in [60].

3.6.5.2 Additional work on GENET for optimisation

The other approach by Boyce *et al* [10] developed for handling CSOPs and PCSPs is similar to the one-stage tunnelling algorithm. The example used to show the optimisation capabilities is the Radio Link Frequency Assignment problem. The general problem consists of a set of frequencies and a set of radio links. Constraints occur because frequencies can have an effect on each other at certain distances. This imposes restrictions on the combinations of frequencies that links which are spatially neighbours can have. A solution is a mapping of radio links to frequencies. The problem can be an optimisation problem considering several criteria. These include: the number of frequencies used, the range of frequencies used and the number of violated constraints (i.e. the problem can be a PCSP).

In [10] to reduce the number of frequencies used, each label has an extra input term. This term is the negation of the number of frequencies that the assignment would have if that label were to be turned on. This extra term, derived from the state of the whole system, can be varied to optimise whatever criterion is desired. This is the main difference between this approach and the 1ST. In 1ST system the extra term for each label consists of the number of variables minus the number of variables that are assigned the proposed value. In [10] the three results where GENET optimises this criterion show that GENET finds the optimal number of frequencies. This is slightly marred by the fact that, in two out of the three problems, GENET can only find the optimal solution in a maximum of 20% of its runs. However, this is not a clear performance indication because in this paper there is no mention of how close the other solutions are to being optimal. Moreover, there is only one other method that it is compared with, Tabu search. In the implementations used in the paper, GENET outperforms Tabu. The success in finding optimal solutions in certain runs was due (at least in the Tabu version) to the way they tackled the following issue. With a single reassignment of frequency to a link there are very few opportunities to remove

a frequency entirely. So even when they tried weighting the cost of using frequencies to be very high the solution was often far from optimal. This was combated by changing the initial solution from a randomly produced one. In a random starting assignment, on average, the number of violations is less than half, but more than half of the available distinct frequencies are used. So the principle of starting with the minimum number of frequencies was used (this could be 1 if all the domains contain a particular frequency). It greatly increases the initial number of violated constraints. However, the system adds frequencies when necessary to reduce violations and so increases the number of frequencies used. Doing this allows the program to find optimal solutions. This work is detailed in [9].

3.6.6 Algorithms derived from GENET

The ideas and principles of GENET were carried forward into a new system which allows greater generality and its basic model incorporates solving CSOPs and PCSPs. This is called Guided Local Search (GLS) [109, 114, 115, 112, 116, 117]. Instead of specifying the objective function as GENET does, GLS leaves it to the implementer. GLS just needs to be supplied with an objective function that maps every compound label to a cost. In each cycle of the algorithm every variable is set to a value that gives the lowest overall cost, breaking ties randomly. This allows sideways moves for each variable but if after a full cycle the total cost has not been reduced then this is treated as a local minimum. The variables are changed in an arbitrary static order. This is a meta-heuristic and so can "sit on top of" other local search methods. This allows such hybrids as the Guided Genetic Algorithm [68].

3.6.7 Conclusions on GENET

To conclude, GENET can be modified to cope with many different tasks and different search strategies can be used. As well as general problems such as graph colouring and random CSPs, GENET has been applied to a few real world problems such as the radio

frequency assignment and car sequencing. However, these changes are not straightforward and require research and experimentation to produce. An overview of some of the above methods are given in [108].

3.7 Summary and Conclusions

The majority of the methods described above are similar in nature; the difference generally lies in the move operator (how it "steps" from state to state) and how it deals with special states such as local minima, plateaus and previously visited areas. When solving real world problems, the move operators and special state operators are often specialising to take in domain knowledge. This is usually done intuitively by an expert in the field. Equally some techniques require the setting of parameters that affect the search. Whilst experimentation and empirical evidence are used to set these, informal heuristics and intuition are often used. These factors may supersede the innate difference in results produced between different algorithms when compared on a specific problem.

The methods here have been described in their basic form and there are always numerous ways of adapting and hybridising them, for example there are several strategies for using methods used in systematic search for local search [105, 62, 102, 128]. There are several studies comparing methods and how the structure of the problem affects the performance of methods, for example [50, 16].

The next chapter will describe how local search techniques shown in this chapter, constraint programming techniques from the last chapter and mathematical programming techniques have been used for producing driver schedules.

Chapter 4

Review of driver scheduling techniques

4.1 Introduction

Early bus driver scheduling systems were heuristic based and limited in their usability. Many were specific to individual bus companies and the techniques used were not directly transportable to other companies. Often a large amount of manual intervention was needed. Some systems were little more than validators. They just checked the shifts and schedules the manual scheduler produced. This was useful but could not be counted as automated scheduling. Later, as computer power increased, the systems could take advantage of mathematical programming. Section 4.2 will describe some early heuristics and in Section 4.3 we will progress to the later mathematical systems. In these two sections we will restrict our review to examples of approaches successful on commercial bus driver scheduling problems. In Sections 4.4 and 4.5 we come to recent work. Here we will broaden

our scope to include theoretical and potential approaches as well as work related to bus driver scheduling. To open this Chapter we will introduce the mechanics of the problem.

Bus driver schedules are designed to ensure that every bus has a driver at all times. Drivers work on pre-planned shifts, each of which obeys certain rules dependent on local legislation and on agreements between drivers and management. Typical types of rule are:

- No shift can exceed a stipulated maximum driving time;
- Every shift must have at least one meal break of a designated minimum duration;
- No part of a shift can exceed a stipulated time on duty without a meal break;
- No shift can exceed a given elapsed time from start to finish (the maximum elapsed time may depend on the type of shift).

In practice, there is usually a variety of further rules. In many countries, including the United States and the United Kingdom, shifts usually consist of *stretches* of work, separated by a meal break. Each stretch may contain one or more *spells* of work, each spell being on a different bus. Drivers can normally join or leave a bus only at designated points (usually one per bus route or line); these *relief points* may be either intermediate or terminal points. We call the times at which buses are scheduled to pass relief points, *relief opportunities*. We may represent the work of a bus throughout a day as a series of relief opportunities linked by indivisible *pieces of work*, each of which must be covered by a driver. A shift therefore consists of two or more spells, each starting and ending at a relief opportunity and consisting of a number of consecutive pieces of work.

Figure 4.1 shows three buses with just the information required for driver scheduling. The solid lines represent the work done by each vehicle; where a relief opportunity occurs, the time and location are shown (here, D is the depot and L is Leeds city centre). The driver assigned to the first bus drives it until the bus returns to the depot. The dashed line shows this spell of work. The driver then has a meal-break and following that takes over the second bus at 1015 in Leeds city centre, from the driver who has already driven that

bus from 0622 onwards. The second driver in turn takes a meal-break and then takes over the third bus from its previous driver, and so on.

4.2 Early heuristic methods

The early methods used for driver scheduling were heuristic based. This was because there was not the computing power to use mathematical solvers. Many of the approaches have similarities. They construct an initial solution using a heuristic process and then make limited alterations to it to try to improve the schedule.

4.2.1 RUCUS/RUCUS II

RUCUS (RUn CUtting and Scheduling) [4, 75, 74] is an example of a system that generates a initial schedule and then heuristically improves it. It first creates single spell shifts and then two spell shifts, after this process any remaining pieces of work that cannot be allocated to shifts are left as short overtime spells. This limits the use of the system and is a reason for RUCUS's demise, as a lot of companies do not use overtime and even if they do they try to restrict it. Further, it is generally inefficient to leave out "difficult" work in this way. Once the initial solution is created the system then uses local search moves to attempt to improve the solution. It either swaps some pieces of work covered by

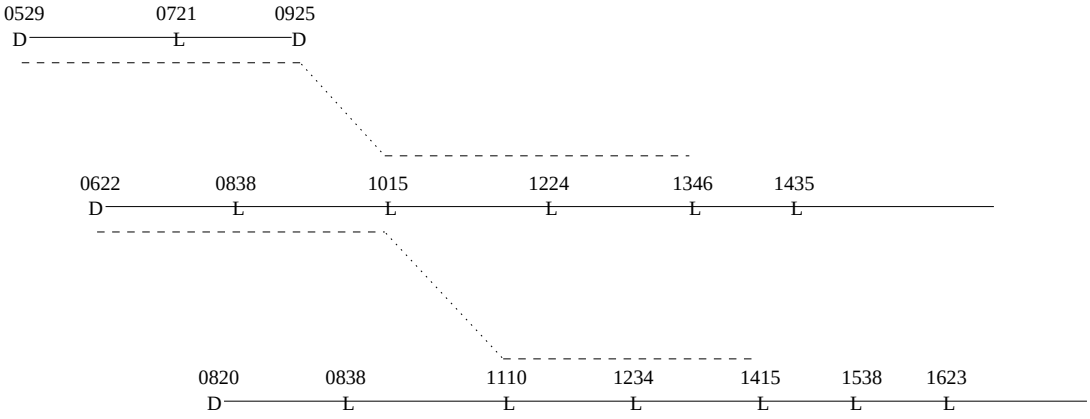

Figure 4.1: A fragment of vehicle schedule showing possible chosen shifts

one shift with pieces of work from another shift or it moves selected relief opportunities forward or backward. There is then a repair procedure which attempts to fix any shifts that have become invalid due to the changes. However, there may still be invalid shifts left in the final schedule and so manual intervention may be needed.

4.2.2 Other heuristic systems

HOT and HOT II (Hamburg Optimisation Techniques) [54, 21, 111] start by trying to form good shifts, one at a time, for each morning bus, and then each evening bus. Any work which is not treated in this process is formed into partial shifts, which are then combined into full shifts by a variant of the Hungarian Algorithm. There is little improvement done to the schedule once it is constructed. Sometimes it may leave unscheduled pieces of work. However, it has been used in several German bus operations. It is believed that it is no longer widespread in use.

TRACS is a heuristic system with a few differences from those already described. This system was developed under the premise that an initial poor solution cannot be altered into a good solution by heuristic improvements. One reason why this may have been true was that development of this system started in 1967 and so the modern local search techniques were not available. A poor solution would be a poor local minimum in the search space and would take several un-improving moves to get to a stage where it could be significantly improved. The heuristics used at this time in driver scheduling tended to use only improving moves and so a schedule could not be greatly enhanced. So while TRACS did do heuristic improvements, similar in nature to RUCUS, it would first concentrate on producing as good an initial solution as possible. This would take a lot of effort working with a bus company to get heuristics specific to the company working, and this process would have to be altered, often substantially, to move the system to a new company. Subsequently, a system, COMPACS, was developed by a commercial company. COMPACS retained the initial solution generation phase of TRACS, but not the improving moves. It could also be used as an interactive scheduling tool and would validate shifts as the

scheduler wrote them.

4.3 Integer linear programming methods

When research into driver scheduling was first undertaken in the 1960s, all practical problems were too large for a mathematical approach using the available technology and methods. To this day, a pure general purpose mathematical approach would still be inadequate to solve practical driver scheduling problems of value. Heuristic reductions are needed and great effort must be put into their development.

4.3.1 Mathematical model of set partitioning and set covering

From the point of view of driver scheduling, the vehicle schedule consists of a set of pieces of work to cover $I = \{1, \ldots, m\}$. We can then produce a large set of possible shifts $S = \{S_1, \ldots, S_n\}$. Each shift covers a subset of the pieces of work ($S_j \subseteq I$ for $j \in J = \{1, \ldots, n\}$). The shifts have an associated cost $c_j > 0$. What we want is a subset of shifts J^* that together cover all the work. This can be written as

$$\bigcup_{j \in J^*} S_j = I \tag{4.1}$$

where $J^* \subseteq J$.

If equation (4.1) holds, then J^* is said to be a cover of I. If equation (4.2) below also holds then J^* is called a partition of I.

$$j, q \in J^*, j \neq q \Rightarrow S_j \cap S_q = \emptyset \tag{4.2}$$

i.e. no piece is covered by more than one shift.

We wish to produce a schedule which has the minimal total cost ($\sum_{j \in J^*} c_j$) i.e. uses the

	1	.	.	.	m
S_1	a_{11}	a_{21}	.	.	a_{m1}
S_2	a_{12}	.	.	.	a_{m2}
.					
.					
.					
S_n	a_{1n}	.	.	.	a_{mn}

Table 4.1: The set partitioning problem

minimum number of shifts. So now we can define our set partitioning problem as the Integer Linear Program (ILP):

$$min\ x_0 \quad = \quad \sum_{j=1}^{n} c_j x_j \tag{4.3}$$

subject to:

$$\sum_{j=1}^{n} a_{ij} x_j = 1, \qquad i = 1, \ldots, m$$

$$x_j = 0, 1 \qquad j = 1, \ldots, n$$

where:

$$x_j \quad = \quad \begin{cases} 1 & \text{if } j \text{ is in the partition} \\ 0 & \text{otherwise} \end{cases}$$

$$a_{ij} \quad = \quad \begin{cases} 1 & \text{if } i \in S_j \\ 0 & \text{otherwise} \end{cases}$$

This problem can be represented as a matrix, as shown in Table 4.1. The rows are the shifts and the columns the pieces of work. When selecting a set partitioning solution we select the minimum number of rows where the sum of each column of the selected rows will be 1.

Often in commercial driver scheduling packages the problem is formulated as a set covering or a set partitioning problem. However, there are often extra features added. For example, side constraints may be imposed to restrict certain types of shifts. The other alteration to the formulation is the incorporation of optimising the number of shifts as well as the cost.

In commercial system a set covering approach is often adopted over a set partitioning one. This is because as a set partitioning problem there will not always be a solution. In contrast to this the set covering formulation is guaranteed to have a solution. In principle, restrictions such as those on depots in train driver scheduling can negate this guarantee but in practice it works as long as an appropriate number of generated shifts cover each piece. In train driver scheduling it is often the case that several depots are in use. The distance between these depots can be so great that provision has to be made to return the drivers to their own depot at the end of a shift. This may mean that drivers need to travel as passengers and systems often cope with this by including the passenger travel in the shift as if the driver were actually driving it. It is then up to the manual scheduler to decide which driver should actually driver the train.

To increase the likelihood of finding a set partitioning solution we would need a much larger supply of possible shifts. This would increase the search space. Nevertheless, the advantage of the set partitioning formulation is that it will produce a schedule with no overlapping drivers (*over-cover*). This is preferable as over-cover creates unproductive time for a driver. Even though the set covering formulation produces over-cover we can reduce the amount of over-cover. For example, in TRACS II some of it can be removed manually or interactively by altering shifts at the end of the process.

4.3.2 TRACS II

The University of Leeds has a long history of driver scheduling research. Its first system for forming driver schedules was a heuristic one outlined above called TRACS [82]. Later a mathematical system called IMPACS was developed in the 1980s. IMPACS is now superseded by TRACS II. This new system has been generated with many train driver

scheduling features in mind but retains the ability to produce bus driver schedules. In this Section we will describe the model that this system uses. We will concentrate only on the newest version of the program. Parts of this system are utilised in the methods that have been generated for this thesis.

4.3.2.1 TRACS II model

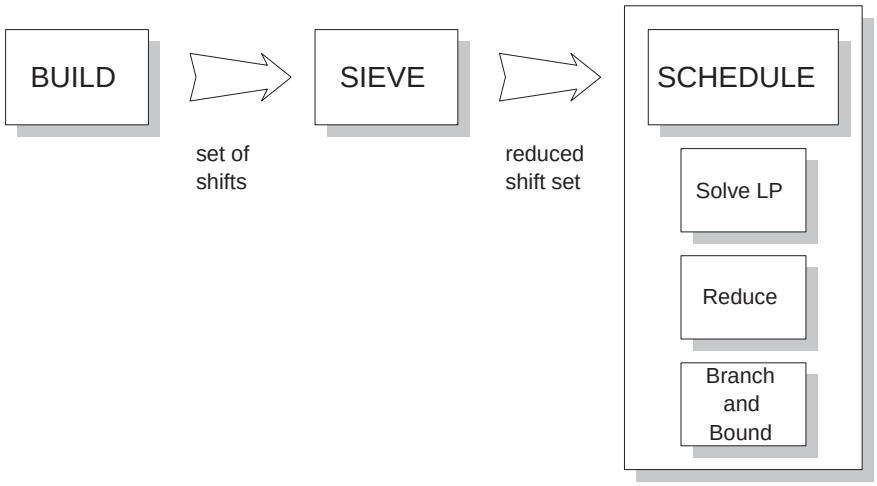

Figure 4.2: TRACS II components

4.3.2.2 Selection of relief opportunities

The IMPACS suite of programs contained a program called SELECT which tried to reduce the size of the problem by removing certain ROs. Unfortunately, this type of reduction can degrade the solutions produced by TRACS II. With the recent improvements of TRACS II allowing it to solve problems of larger sizes the SELECT program is now never used. However, sometimes ROs are removed manually by skilled users when problems are too large. Work described in Section 4.4.7 is an attempt to replace the dated SELECT module.

4.3.2.3 Duty generation

The BUILD program generates a large set of valid shifts. It is described here but further description can be found in [66] (although that paper relates to the conceptually similar rail driver scheduling problem). The first priority of this program is to produce only shifts that are valid. However, there are many more aspects it has to consider. If too many shifts are generated the problem may become too large for the mathematical solver to find a solution in a reasonable time. On the other hand, omitting important shifts can be detrimental to the efficiency of the final schedule produced. So the BUILD process tries to only produce "good" shifts. This is a task that takes several heuristic rules because the ultimate decider as to whether a shift is good or not depends on how it combines with other shifts and this cannot be found out until the problem is being solved.

The BUILD process starts by generating a large number of *spells*. Rules apply to the minimum spell length so as not to produce spells which contain inefficiently little amounts of work. These spells are then combined where appropriate into *stretches* of one or two spells. Stretches also have a minimum length so as to prevent inefficiencies, but they also conform to rules governing their maximum length, which is usually the maximum time a driver can work without a meal break. These *stretches* are then combined to form shifts of up to four spells. Between consecutive spells of work the driver has *joinup* time to get from an RO where one spell finished to another RO where the new spell starts. The other possibility is that the driver will have a meal break, if the time is sufficient. Rules are applied from legal, union and company practice, such as minimum meal break length and maximum driving time. As well as these common sense considerations it includes such rules as not producing shifts that contain a spell on a bus followed by a joinup and then a spell continuing on the same bus. Once the shifts are generated, shifts that are seen to be obviously "poor" shifts compared to others are removed. For example, as shifts containing a high number of spells are not usually desired, three spell shifts that are inefficient compared with two spell shifts containing a substantial portion of the same work are removed.

There are many different types of shifts. Morning, evening, day, overtime and split. Each may have its own regulations. These are governed by parameters which have to conform with differing bus company regulations.

4.3.2.4 Reduction of the set of Duties

It is sometimes the case that BUILD produces more shifts than the mathematical solver in SCHEDULE can handle, or more than is necessary to obtain a good solution. The original IMPACS version used a process called EVEN. This operated by removing shifts that covered pieces of work that were also covered by many other shifts. TRACS II uses a different process, called SIEVE.

SIEVE initially removes shifts that are duplicates of other shifts. Next, SIEVE asks the user to give a target number of shifts to remain after the process. SIEVE then ranks each shift according to: a measurement of its cost effectiveness and the least and average number of other shifts covering the pieces of work that the shift does. SIEVE then starts removing the lowest ranking shifts, as long as this does not leave work uncovered, until the target number are remaining. At certain stages SIEVE recomputes the ranks of remaining shifts, to reflect the fact that low ranking shifts may become critical after those of lesser rank are removed. The user then gets to reinstate shifts if they feel these shift's cost effectiveness is too high for them to be removed.

4.3.2.5 LP relaxation

The aim of the mathematical solver is to select a set of shifts from the large set of potential shifts. Several criteria are to be optimised in this process, the most important usually being the number of shifts. TRACS II takes all of these considerations into a single optimisation criterion. In this description we will describe the newest versions of components that are incorporated in SCHEDULE. SCHEDULE is based on ZIP [89] and still retains much of its principles.

As well as the set covering constraints there are sometimes user defined side constraints. These are set to limit the number of different types of shifts.

The first part of the process is to relax the integrality constraints to allow fractional solutions. The solver then uses an initial solution to start the optimisation process. The initial solution was originally produced by selecting still uncovered pieces of work one at a time and then choosing a shift to cover it that minimises the following function

$$\frac{C_j}{NU_j} \tag{4.4}$$

where C_j is the cost of the shift and NU_j is the number of currently uncovered pieces of work covered by shift j. However, a new initial solution method was developed by Willers [123], suggesting the shifts should be selected by:

$$Max \sum_{i=1}^{M} x_{ij} L_i \tag{4.5}$$

$$\text{where: } x_{ij} = \begin{cases} 1 & \text{if shift } j \text{ covers the currently uncovered piece of work } i \\ 0 & \text{otherwise,} \end{cases}$$

$$L_i = \text{duration of workpiece } i.$$

There is not much difference between the quality of the initial solutions produced by these two processes, although the second process is on average better. However, either process will provide a starting solution which will lead to an optimal solution to the relaxed LP.

The solver used for the relaxed LP is the dual steepest edge approach [59]. If there is a large number of potential shifts, a column generation process developed by Fores [36] is used. Once the problem is solved a new constraint is added which increases the (possibly fractional) total number of shifts used, up to the next highest integer. This will be the lower bound on the number of shifts in the optimal integer solution. The model is then

re-solved using the dual steepest edge approach.

4.3.2.6 Branch and Bound

Smith [100] introduced a method of greatly cutting down the number of shifts and the number of relief opportunities that go into the branch and bound method. This process is called REDUCE. It removes many shifts from the search by only using shifts that start or end at an RO that is used in the LP solution.

Once this has been done the process continues into the branch and bound phase. The process branches on relief opportunities, this means that nodes of the search tree correspond to ROs and have two branches; either the RO is used (1) or not used (0). This process uses the relaxed LP solution to form fractional values for each RO (the details of this are found in Section 5.5). These values are used to choose which branch of a node to explore first. The algorithm explores the 1 branch if the fractional value is closest to 1, and the 0 branch otherwise. The process could just branch on the shifts, i.e. each node would correspond to a shift and each branch to whether the shift is used or not. The reasons why it does not do this are discussed in Section 5.5. The process will run until a solution is found with the minimum number of drivers or it has explored 500 nodes. If a solution is found the process tries to further optimise the solution to reduce the overall cost of the shifts in the schedule, until 500 nodes have been explored.

4.3.2.7 TRACS II summary and results

TRACS II is incorporated in a commercial system that has been successfully installed in several transport companies. An example of a problem that is near the upper bound of the size of problem that TRACS II can deal with using the column generation enhancements is a problem with 53297 potential shifts, 976 pieces of work and 195 shifts in the final solution. Some problems have greater numbers of potential shifts entering ZIP, and similarly greater numbers for pieces of work and shifts in the final solution. However, this problem is overall

one of the largest problems solved without decomposition.

Figure 4.3 illustrates the process that is used by TRACS II to produce a schedule. The initial stages are to remove potential ROs. This is done in several ways; by hand, possibly in the future a new procedure by Layfield *et al*, described in Section 4.4.7, and Smith's reduction [100] described in Section 4.3.2.6 is used just before entering the branch and bound phase. The final stage is to decide on the ROs that are to be used in the final selection. This is done by the branch and bound process described in Section 4.3.2.6. Once this is done the shifts to be used in the final schedule are virtually set.

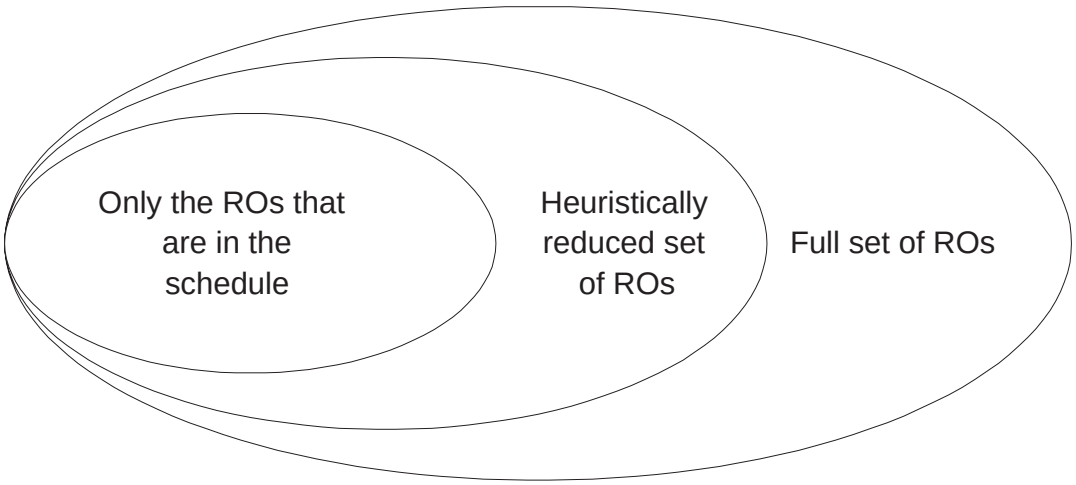

Figure 4.3: The different levels of RO selection. ROs can be removed either intuitively or heuristically to produce a reduced set or all ROs that are not in the final schedule can be removed.

4.3.2.8 Scheduling side issues

Although the driver scheduling problem is often modelled as a straightforward set partitioning or set covering problem there are sometimes further restrictions. In this section we will explain three such restrictions and how they relate to TRACS II.

One restriction arises because often certain types of shifts are undesirable. For example a split shift, where a driver will do a stretch of driving in the morning followed by a long break and finally do a stretch of driving in the evening. These shifts are used because

there is often a peak in the number of buses on the road during the morning, for people going into work, and the evening for their return home. However, the spans of these shifts are long and so many drivers dislike them and unions sometimes make agreements with management to restrict the number of this type of shift in a given schedule. This problem has been effectively modelled in mathematical programming by side constraints.

A difficulty has arisen for the TRACS II system in its development for solving train driver scheduling problems. These problems, unlike bus driver scheduling ones often, have many depots, and due to the often large distances between them, provision has to be made for drivers returning to their own depot. Further restrictions apply because often there is a limit on the number of drivers that can come from each depot. This causes problems in a few ways. Normally a schedule can be found with the number of drivers that is the same as that of the lower bound given by solving the relaxed LP problem. However, it is possible that more drivers are needed when multiple depots occur. The reason for this has not been proven but it may be due to the fact that often in the relaxed LP solution a piece of work will be fractionally covered by shifts from different depots and when the integer solution is derived both shifts will need to be used. Another problem is if SIEVE needs to be used, sometimes shifts that cover similar work but come from different depots would be removed by the SIEVE process but have to be retained because of the depot restrictions. This does cause a small increase the size of the problem.

There is one problem with the set partitioning/covering model that has not been tackled in the TRACS II suite. This is windows of relief opportunities. Often vehicles arrive at a relief point and remain there for several minutes before moving on, the actual time varying considerably. Under some operating agreements a driver change can be made at any time between the vehicle's arrival and departure. However, in a set partitioning/covering model when generating shifts, a specific point and time (an RO) is needed to create shifts. To have an RO for every minute a vehicle stands at a relief point would increase the problem size by an unacceptable amount. At present the RO time is normally taken to be when the vehicle arrives at the relief point. Unfortunately, due to union agreements this might mean that shifts that could in theory be allowed are not generated. This could be because

it may affect some issues such as maximum time before a meal break. It is possible that a driver starting work when the vehicle arrives at a relief point might have to drive for too long before they can be replaced at a time and place that makes an efficient spell.

4.3.3 HASTUS

HASTUS [7] is a suite of programs that contains programs for crew scheduling as well as for bus scheduling. The HASTUS crew scheduling component is broken down into two systems, HASTUS-micro and HASTUS-macro. HASTUS-macro provides an initial solution and HASTUS-micro generates the final solution. HASTUS-macro uses linear programming to generate a pseudo-schedule that provides an estimate of the number of drivers that are needed. The pseudo-schedule is built by pseudo-shifts, which are generated using Pseudo-ROs, which are simplifications of the ROs; this is done by just cutting the day into user defined time slots. The pseudo-schedule is also used by HASTUS-micro to produce a final schedule by using it to produce real shifts that relate as close as possible to those in the HASTUS-macro solution.

`CREW_OPT` [27, 26, 88] is a system that uses column generation to produce schedules. Initially it could only be used for small scheduling problems but more recent work [88] suggest it has potential to replace the older HASTUS components.

HASTUS has been used widely in transport scheduling as it provides a graphical user interface and a system that deals with all the scheduling issues: bus, driver scheduling and rostering.

4.3.4 EXPRESS

EXPRESS [34, 35] is a bus driver scheduling system developed for a company in Christchurch, New Zealand. This is an example of a method that uses a set partitioning formula. However, during the search process the strictness of the model is diminished by

the addition of slack variables. It then uses a version of the original ZIP [89] program that components of SCHEDULE in TRACS II are based on. The branching model is slightly different from the one used in TRACS II, in this system the branch and bound algorithm branches on the pieces of work (constraint branching) rather than the relief opportunities. Branching on ROs was found to be a superior search strategy by Smith [100].

4.3.5 Air crew and bus driver scheduling compared

Much of the work done on constraint programming for solving set partitioning problems has been done on problems derived from air crew scheduling [48, 81, 87]. There is a set of benchmarks for these in [2, 53]. However, the terminology differs between bus driver and air crew scheduling and from company to company. The equivalent of shifts in air crew are usually called rotations or pairings. The equivalent of pieces of work are usually called flight legs. More importantly the internal structure of the two types of problem can be very different. There tends to be a lot more pieces of work in bus schedules than flight legs in aircraft schedules. This is because in air crew schedules a flight leg may last many hours, whereas in bus driver scheduling a piece may be as short as 10 minutes. For this reason, if we generated all possible shifts, even small bus schedules would become impractical to solve. Thus, we have to restrict the number of generated shifts, by using heuristics so as not to generate shifts that are thought to be "poor" in some sense e.g. they cover a small amount of work. However, this may lead to pieces of work that cannot be covered without shifts overlapping (over-cover) and so in our generated shift set we may not have a set partitioning solution.

4.4 Constraint programming methods

Constraint programming approaches for producing full crew schedules have been almost exclusively restricted to air crew scheduling. Furthermore, most of them depend heavily on the use of LP solutions to guide variable and value ordering. Two exceptions to this rule

are the systems described in Sections 4.4.5 and 4.4.4. Some of these methods have been mentioned in Section 2.8 to illustrate points about comparing systems. In this section a more detailed account of these systems will be given.

4.4.1 Guerinik and Caneghem

Guerinik and Caneghem [48] devised a constraint programming approach which used mathematical programming (MP) as a guide to solve the set partitioning problems derived from air crew scheduling in [2]. The system starts by applying mathematical reductions on the set partitioning problem as a preprocessor phase. These will be further discussed in Section 5.4.

This approach models the problem using the rotations as the variables, in the same way as the ILP model does. The variables are ordered according to their corresponding fractional value's closeness to 1, the closest first. The value first attempted for each variable is 1. So while there are no fails the indication given by the values of the relaxed LP solution is consistent with the choices made. However, when a fail occurs a variable will attempt the value 0 and by so doing the relaxed LP solution will no longer be an accurate guide and therefore the relaxed problem will be re-solved. The system does not perform as well as a pure mathematical programming approach that was presented by Hoffman and Padberg [53].

4.4.2 Rodosek *et al*

Rodosek *et al* [87] produced a general way of combining mathematical programming and constraint programming. When the system is used to solve a problem it first solves the relaxed problem by an LP solver. It then uses this to order the variables, according to their closeness to 0 or 1 (closest first). It then chooses the nearest integer value to the fractional value as the first choice for each variable. Whenever there is a fail a new value is tried and the relaxed problem is resolved with the existing assignments and the new

assignment set. In this way the fractional values are affected by previous decisions and so become a more accurate prediction of what the final integer values will be.

One of the problems that was used to test this system was an instance of the air crew scheduling problem. It was the smallest one from the set given in the ORlib [2]. To compare their hybrid system they produced a pure constraint programming approach. This CP approach used the rotations as variables, as with the Guerinik and Caneghem model. They also produced a pure mathematical programming approach using CPLEX [18]. This MP approach produced the optimal solution to the set partitioning problem that they showed in a much shorter time than the CP approach. The hybrid approach took longer than the MP approach, but much shorter than the CP. So the hybrid approach did not seem to get anything useful from the constraint propagation, in fact, it was detrimental as it slowed the process down. This may mean that with the Guerinik and Caneghem approach the LP solver is also doing almost all of the work in solving the problem. The strong point of the system is that on the range of problems shown it usually did better than one of the pure CP or MP approaches.

4.4.3 Müller

Müller [81] produced a pure constraint programming system for solving the air crew set partitioning problems from ORlib. The system applies a pre-processor to make several mathematical reductions on the problem size, in a similar way to Guerinik and Caneghem. However, Müller uses one reduction which is different from the ordinary mathematical ones. This one first orders the rotations with the lowest cost ones first. It then goes through and replaces any single shift that can be replaced by a set of shifts which cover the same flight legs but have lower combined cost. This reduction would not be useful in a system producing driver schedules, because the desire to reduce the number of distinct shifts in the schedule means that replacing single shifts with multiple shifts would not be a good idea. Besides, in driver scheduling heuristically constructed shifts are unlikely to be able to be replaced in this manner.

The model is then set up in the same way as the pure constraint programming approach by Rodosek *et al* but the constraints are implemented differently. They add what they call index sets to the model. There is one for each element i (see Equation 4.1). These sets hold the indices of the subsets S which cover i. When a variable is assigned a value, 1 or 0, this has an effect on the index sets. If the variable associated with S_j is assigned a value 0 then j is removed from all the index sets. On the other hand, if it is assigned the value 1 all the index sets that contain j are reduced to the singleton $\{j\}$. If any of the index sets are reduced to the empty set then a fail has occurred and backtracking happens. This model of constraints will be further examined in Chapter 5.

This system could solve problems but the size of the problem solvable was much smaller than those systems using mathematical programming.

4.4.4 Darby-Dowman and Little

Darby-Dowman and Little [22] created a simple CP program in 1998 for producing driver schedules aimed at reducing crew costs (not number of drivers). They model the problem in a set covering formulation, with the pieces of work as the variables and the indexes of the shifts that cover that piece of work as domains. The constraints are different from those in Müller [81]. For each piece of work a counter is stored to show how many shifts cover that piece of work. If a variable is set to a value the counter is incremented. Further, if a variable is set to a value, all the variables that have that value in their domain are set to that value if they are not already bound. If they are, their counter is incremented. The counter starts at zero and when a value is chosen for a variable then if there is no value that will keep the counter below three the variable is left unassigned and the piece of work associated is left under-covered. This model allows very little constraint propagation and it is unsurprising that this method produces poor results with large amounts of over-cover and under-cover.

4.4.5 Charlier and Simonis

Charlier and Simonis have produced a new constraint programming approach for producing driver schedules. It is a commercial system designed for North Western Trains. There is little known of the details of the system, the only published material is an abstract [15]. What the system seems to do is generate shifts in a sequential order to produce a full schedule. It is believed to model the problem as a directed graph. Each node is an "activity" (presumably a piece of work). The arcs of the graph represent the possibility of having the two activities associated with the two nodes connected to the arc following each other in a shift. An arc has a weight to indicate how "good" an idea it is to have the implied sequence of activities in a shift. A shift is then generated by a shortest path heuristic. The results are unclear and it is believed that it is not presently being used to produce real schedules.

4.4.6 Yunes *et al*

Yunes *et al* [129] in 1999 developed a hybrid CP/ILP approach for producing bus driver schedules. The ILP approach is used to solve the set covering problem, while the CP approach generates shifts for the problem. The ILP approach is a column generation approach where the set covering problem is solved with a minimal set of shifts. Then shifts are added in to see if the solution can be improved. The CP approach produces these shifts that are added into the search.

The system has been tested on real data from a Brazilian transit company. It has achieved good results on relatively small problems (150 pieces of work, with 19 shifts in the optimal).

4.4.7 Layfield *et al*

Layfield *et al* [69] used constraint programming to produce a component that could slot into the TRACS II system. It would be put before the building phase and would do the

same job as SELECT used to do in the IMPACS version. The goal of the program is to remove relief opportunities that are unlikely to be used in good schedules, thus cutting down the size of the problem because not only will there be fewer pieces of work but if fewer ROs are used the BUILD process will produce fewer shifts.

The program initially looks at the morning part of the schedule. It produces shifts using knowledge of how a manual scheduler might do it. The program puts a limit on how many spells of work each of the buses will be broken up into, so that it does not produce shifts with spells that are too short. It constructs a morning schedule using randomised heuristics to build the partial schedule one shift at a time. It does this several times and then removes the ROs that are not used in any of the schedules. It can also be used to construct a partial schedule for the evening part of the schedule and thus remove further ROs. The process has speeded up TRACS II's solution time in several cases. The cost of the solutions are often slightly higher but sometimes less. The solution can have a lower cost because TRACS II does not produce solutions with optimal cost and stops when it gets to a "good" solution. So when TRACS II uses the cut down version it might come to a lower cost solution than the original before it stops.

4.5 Evolutionary algorithms and other meta-heuristics

4.5.1 Tabu search

Cavique *et al* [12] have used Tabu search [46] to extend and improve one of the methods used in the early heuristics. Their algorithm starts with an initial solution produced using an approach similar to that used by TRACS. The method allows shifts that contain two spells of work or even less efficient shifts that cover single spells of work. The improvement phase then incorporates Tabu search. A move consists of removing a number of inefficient duties, and sometimes their neighbours and then generate shifts to make the schedule whole again. Tabu search is used to ensure that pieces of work that appear frequently in inefficient shifts are given higher priority in incorporating into shifts that contain two

spells of work and so are more likely to be efficient. This is done to try to cover pieces of work that are hard to cover using efficient shifts. The work they did on this Tabu Search approach found that the method quickly improved the solution over the first few iterations but then found it hard to make further improvements. This is possibly because they only concentrate on inefficient shifts and sometimes an efficient shift may have to be changed to make the leap to a really efficient schedule. They also provide another approach that uses a matching technique that does better, possibly because it expands the search, not restricting it to changing inefficient shifts.

These algorithms were developed for the Lisbon Underground. There are several features to note about this operation. There is a maximum of two spells used in shifts. There are also no costs per shift, it is a straight minimisation of the number of drivers. Further, there is only a short amount of driving time in each duty (less than 5 hours) and the drivers can only change at the terminus. These differences from the standard make it hard to judge how the Tabu Search program would work on problems from other companies. It may have the same drawback as the early heuristics in that it would be hard to adapt to different bus or rail operations.

4.5.2 Kwan *et al*

The approach by Kwan *et al* [67] uses a genetic algorithm to produce driver schedules. This work was built on experience of the earlier attempt to do this by Wren and Wren [127]. This system uses the potential shifts generated by TRACS II. It also uses the LP solution produced by TRACS II. In this system a complete representation of a schedule by each chromosome is abandoned to form a concise representation that incorporates the essence of the schedule. This is done by the chromosome being made up of bits for each shift in the LP solution generated by TRACS II. The reason why only these shifts are represented is that empirical evidence has shown that at least 50% (and up to 98%) and on average 74% of the shifts in the final TRACS II solution were in the LP solution. So these shifts make the backbone of the schedule, and once a good combination of these is found it should be

much easier to make a good whole schedule. To make an entire schedule out of these a greedy repair technique is used.

This method has produced schedules for some problems with the same number of shifts as the TRACS II solutions. Unfortunately in other some cases it does not get the same number of shifts, it has one or two more shifts. The strength of the GA method is that it will always find a solution and has found solutions to problems that TRACS II could only solve after they have been decomposed into subproblems. In these cases it has found solutions with fewer shifts than the total number of shifts of the union of the decomposed schedules produced TRACS II. For example, it found a solution with 267 final shifts where the union of the TRACS solutions had 276 shifts.

The technique incorporates the use of any good traits of a schedule to affect the valuation of the schedule for mating. This would mean that schedules that had good parts but were average overall would have a chance of mating. Ideally the mating process would be biased to pass on the good segments of the schedule.

4.5.3 Chu and Beasley

Chu and Beasley have used a genetic algorithm to solve set partitioning problems derived from air crew scheduling problems. The basic model is to have the genes representing n bits where n is the number of columns. Each bit can be 1 for a column contained in the solution or 0 for those not contained. This could lead to very large strings as the size of problems grow. Regardless, this is the model used and a uniform crossover approach is used. The algorithm differs from a standard GA in the way optimisation criteria are dealt with. Each chromosome may or may not give a feasible partition. They note that finding any set partitioning solution is not a trivial task for heuristic approaches. So it is important to drive the solution towards a feasible one as well as trying to reduce the cost of the solution. A standard way to deal with restrictions imposed on a solution is to add a penalty value that is subtracted from the objective value of each solution. However, this could lead to loss of good parts of solutions with high objective value but also with a high

penalty value. To overcome this they have added a dual optimisation criterion with one measure being the cost and the other the feasibility of the solution. The choice of parents was then made on a combination of these. The crossover and mutation would often lead to solutions with either large amount of over-cover or undercover. This was solved using heuristic repair. The algorithm was successful on air crew scheduling problems. However, this problem is thought to be easier than the bus or train driver scheduling problem.

4.5.4 Forsyth

Forsyth [38] has applied an optimisation method called the Ant system for producing driver schedules. An Ant system was developed by Dorigo *et al* [30] based on the method ants use to search for food. A simplified version of how ants forage for food is converted into a search algorithm in the following way. In the simplified version the ants set off from a nest in random directions. As they move they leave pheromone trails behind them which slowly evaporate over time. When food is found the ant returns to the nest travelling back with highest probability along its own pheromone trail, thus strengthening the trail. The ants have an in-built bias towards following strong pheromone trails so over time more ants will come across this trail and follow it, strengthening it even more. As there is still randomness in the ants' movements several paths will be made between the food and the nest. However, the shortest path will gain the largest deposits of pheromones, as ants will return along it sooner than ants on other trails.

The ant system for driver scheduling uses the shifts generated by BUILD to create a schedule. Each ant component will create a solution at each iteration. RO's are selected by a probabilistic heuristic and then the ant chooses a shift from the set that start at that RO. This is repeated until all the work is covered. As the system progresses iteration by iteration the good parts of solutions are more likely to be followed (i.e. good combinations of shifts are selected) and so over time the solutions improve.

This method does not produce results comparable to the TRACS II system.

4.6 Summary

This Chapter contains a brief introduction to the area of driver scheduling. For further reading, an overview paper given by Wren and Rousseau [126] is a good round up of work done to that date, however this was written in 1993 and much research has been done since then.

Almost all and possibly all of the early heuristics are no longer used in present commercial organisations. They tended to be hard to adapt to new conditions and once that was done they needed extensive manual intervention to produce schedules. Systems that use mathematical programming such as TRACS II and HASTUS have taken over from these early systems. They still sometimes need adaptation to new conditions and some manual intervention but are an enormous improvement on the initial heuristic methods. There have been new heuristic methods tried, with the genetic algorithm by Kwan *et al* being the most promising, but none have achieved the quality of the mathematical programming approaches. This is not surprising as over 30 years of experience has been put into the TRACS II system. However, there is room for improvement in two features. TRACS II cannot prove that the solutions it produces are optimal, so it is possible that better solutions exist and can be produced. Further, the flexibility of the system can be improved, as shown by the side issues described in 4.3.2.8. Before either of these issues can be tackled by a new system a basic process needs to be produced. Then it can be further developed to improve solution quality and investigate to see how to incorporate the side issues. The next two chapters will detail the development of two systems to produce a basic process for producing driver schedules.

Chapter 5

Driver scheduling using constraint programming

5.1 Introduction

This Chapter describes a systematic constraint programming approach to solve the driver scheduling problem. It starts with a model that could be used on any set partitioning problem and it is explained in the Summary the exact parts that could be used to solve general set partitioning problems. Domain knowledge is incorporated to develop a new model. Much of the work in this Chapter has previously been published by Curtis *et al* [19].

5.1.1 Set partitioning or set covering?

The first question to ask in developing a new system is whether we should use either of the present standard formulations, which are set partitioning and set covering, or should we use a different representation. Early heuristic methods did not use a set partitioning/covering approach, they generated shifts as needed. However, this means that all the union agreements and other restrictions need to be built into the solver and hence the solver has to be altered every time the conditions are altered. Further, the solver may be too domain-dependent and be poor or useless on problems with different regulations. Charlier and Simonis [15] developed a system using constraint programming where the schedule is built up as shifts are generated (see Section 4.4.5). However, as mentioned the details are unavailable at present. What information that is available shows that the system has only been produced for one organisation, North Western Trains, but it is unclear if it is in operation. For the reasons just given it is unlikely that this system would be easily adaptable for use with other rail organisations.

We have opted for a set partitioning/covering formulation. We use the shifts generated by the TRACS II component BUILD (see Section 4.3.2.3) to provide the initial pool of shifts to select from. This means that our program needs no knowledge of what constitutes a legal shift. The program does not use the knowledge of how shifts were built to construct a schedule. This makes it (in principle) independent of any changes in how the shifts are constructed. The next decision was which of these two formulations, set partitioning or set covering to choose from. As stated in Section 4.3.1 the hindrance with a set partitioning formulation is that there may be no solution to the problem with the current set of shifts or that to find a solution to a problem, the solver would need a greater pool of potential shifts than if a set covering formulation was used. However, it is difficult to work with a set covering formulation in constraint programming, because the decision to include a shift in the schedule leads to no constraint propagation, whereas in set partitioning once a shift is chosen we can remove all other potential shifts that cover any pieces of work in common with the chosen shift. This propagation is needed to guide the search so that it

does not use unnecessary shifts. There has been a program devised by Darby-Dowman and Little [22] that used a set covering approach but it did not perform well and one of the reasons for this is probably due to the lack of constraint propagation (see Section 4.4.4). Therefore, we choose to use a set partitioning formulation.

To illustrate our algorithm's ability to solve problems we have, in our results, only included instances that we know to have a set partitioning solution within the available set of shifts.

5.2 The Models

How a problem is modelled as a constraint satisfaction problem can greatly affect the performance of the algorithm. We will discuss two models of the set partitioning problem as a constraint satisfaction problem and their advantages and disadvantages. We have implemented the second model, as well as an extension to it that greatly improves performance. This is described later in the Chapter.

5.2.1 The first model: shifts as variables

The most obvious representation is a straightforward translation of the mathematical programming model described earlier in Section 4.3.1. The shifts are the variables, with a binary domain [0,1], where 1 means that the shift is used in the solution and 0 means the shift is not in the solution. This is the model chosen in the papers of Guerinik and Caneghem [48], Rodosek *et al* [87] and Müller [81].

The constraints follow directly from the set partitioning formulation. For every piece of work one, and only one, shift variable that covers that piece can be set to 1. So when we set a shift variable to 1 we set all the other variables of shifts that cover a common piece of work to 0. The number of possible assignments of values to variables in this model is 2^n, where n is the number of shifts: this is an indication of the complexity of the model.

One drawback of this method is that, when it is decided to use a shift i.e. assign the corresponding variable the value 1, a powerful decision is made, removing many other possible shifts from the search space. When the decision is made not to use a shift, it makes very little difference as there are probably several other available shifts that cover the work in question. Later we will discuss how to alleviate this "all or nothing" choice.

5.2.2 The second model: pieces as variables

The following is the formulation investigated in this Chapter and implemented using ILOG Solver version 3.2 [57, 58]. The variables represent the pieces of work P_i (where $i \in I$, the set of indices of the pieces of work). The domain of each variable, D_i, is the set of indices of the shifts that cover piece of work i ($D_i \subseteq J$ where J is the set of indices of the shifts). If P_i is assigned the value $j \in J$ then in the schedule, piece i is covered by shift S_j.

In this formulation the number of possible assignments, $\prod_{i=1}^{m} |D_i|$, is less than the previous model. This representation automatically ensures that in the solution all pieces of work have a shift covering them, because every variable must have a value.

Suppose variable P_i is assigned the value $j \in J$. Then the i^{th} piece of work will be worked by shift S_j. This implies, because of the set partitioning formulation, that all pieces of work covered by S_j will be performed by shift S_j. So for any other piece of work such that $j \in D_k$ we must have $P_k = j$. This gives the constraint:

$$(P_i = j) \iff (P_k = j)\ \forall i, k \in I \text{ such that } D_i \cap D_k \neq \emptyset,\ \forall j \in D_i \cap D_k \tag{5.1}$$

So if piece i is assigned shift j then piece k will be assigned shift j and vice versa. This can be expressed easily in Solver, as an equality constraint. A second constraint is added to propagate efficiently the effect of choosing values in a set partitioning formulation. If one piece variable has a value removed from its domain e.g. the variable is assigned another

value, then the removed value must be removed from the domain of all other variables. If a shift is not used by one of the pieces of work it covers, it cannot be used by any of the others. For this a new constraint was developed using the Solver constraint template. This is called the *rem* constraint.

These constraints could be applied (posted) to all piece variables with shifts in common. However, there is a way of reducing the number of constraints. This is done by using the following: for each shift, the constraints are posted between only one of the pieces which is covered by that shift, chosen arbitrarily, and all the other pieces that are covered by that shift, rather than between all pairs of pieces covered by the shift. Then the effect propagates through these pieces.

This model has the same drawback as the last model in the "all or nothing" choice of shifts. We will see in the extension to this model, described later, how we can use the structure of the driver scheduling problem to overcome this.

Although Müller's approach [81] described in Section 4.4.3 was independently developed it relates to the method used here. It uses shifts as the variables, with a binary domain and not pieces of work as we do. However, to implement the constraints, Müller employs what he calls index sets which are sets of indices of the subsets S (in our case these would be the shifts). This definition matches our definition of the piece variables, because the domains of our piece variables are the indices of the shifts covering them. Müller then applies the same two set partitioning constraints as we do on these index sets. During the search the choices are made on the binary subset (shift) variables and the effect is propagated to the index sets. If an index set becomes empty then backtracking is instigated. So the only important difference between our second approach as described so far and Müller's algorithm is that Müller's assigns values to the binary subset (shift) variables whereas ours are assigned to the piece variables. The benefit of processing the variables in our way will be made clear in the next section.

5.3 The Search method

Solver's standard backtracking algorithm which maintains arc consistency is used (See Section 2.3). We customise the search by variable and value ordering heuristics.

A useful variable ordering that has been applied in many constraint programming applications is to choose the unbound variable with the smallest domain (see Section 2.4.1). We can see that assigning values to piece variables allows a more natural use of this ordering than using shift variables, because these would all have binary domains.

In the first version of the system we used a branch and bound method to minimise the number of shifts. In this approach, once a solution is found, we try to find a solution with a lower objective value. When no new solution can be found, the existing solution is an optimal solution.

The objective here is to reduce the number of shifts used, so when a solution with $n+1$ shifts was found the algorithm applied a constraint that no more than n shifts could be used and started a new search. This constraint only had a propagation effect when close to n shifts had been assigned. So if we started the new search with no shifts assigned the algorithm had to assign nearly n shifts to being used before the constraint could act. To improve on this, when a new search was started, the old assignment was used as a starting point (we call this "restart from existing solution"). Table 5.1 illustrates how this strategy and using the smallest domain ordering affected performance. The problems come from three different bus companies: Reading (**r1** to **r4**) [125], CentreWest Ealing area (**c1**, **c1a**), the former London Transport (**t1** and **t2**). The problems have differing regulations and features (e.g. urban and short distance rural bus schedules can have very different features). The size of the CSPs representing these problems is given. Note that **r1** and **r1a** are the same problem, but have different numbers of generated shifts and **c1**, **c1a** similarly.

The program is run on a networked Silicon Graphics O2 workstation. It is stopped af-

Instance	t1	r1	r1a	r2	t2	r3	c1	c1a	r4
pieces	24	53	53	54	125	160	186	186	203
initial shifts	77	2503	4273	3001	3015	19091	3829	7543	2484
size	271	12k	21k	13k	17k	214k	27k	53k	17k
opt # shifts	7	11	11	14	19	16	26	26	25
rn:									
best result	7	15	18	18	21	om	n/a	n/a	31
fails	5	2053	544	432	4972	n/a	>10k	>10k	4510
time (secs)	0.06	299	148	62	612	n/a	>590	>7.4k	268
fails to prove opt	168	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
time to prove opt	0.61	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
rn sd:									
best result	7	18	18	19	22	om	30	31	31
fails	2	231	1543	1809	6192	n/a	1790	96	9569
time (secs)	0.05	45	416	325	938	n/a	191	64	807
fails to prove opt	168	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
time to prove opt	0.49	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
rs:									
best result	7	15	18	18	21	om	n/a	n/a	31
fails	5	2018	485	299	4891	n/a	>10k	>10k	4510
time (secs)	0.06	295	76	592	595	n/a	>900	>7.6k	273
fails to prove opt	163	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
time to prove opt	0.52	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
rs sd:									
best result	7	18	18	19	22	om	30	31	31
fails	2	223	1437	1796	4838	n/a	921	96	9558
time (secs)	0.052	38	361	301	725	n/a	94	65	810
fails to prove opt	119	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
time to prove opt	0.45	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 5.1: Results on data from several bus companies using different regulations. sd = smallest domain ordering, rs = restart from previous, rn = restart from no assignments, opt = optimal, om = machine ran out of memory

ter 10000 fails (10000 backtracks) and we use the number of shifts in the best solution obtained, the number of fails, and the time taken, as the performance criteria. The number of fails shows us how many times our program backtracks. We can see from the table that, using these basic methods, only for the **t1** problem can an optimal solution be found. (Throughout the Chapter optimal means here, the optimal number of shifts for the set of potential shifts after heuristic reductions.) If no solution, or no optimal solution, has been found we put n/a in the appropriate column. If we restart from an existing solution each time a new one is found, in all cases the number of fails is reduced. Using the smallest domain ordering generally increases the number of fails to find the best solution found. However, it does enable us to find a solution for the **c1** and **c1a** problem that we could not find an answer for otherwise.

5.4 Reductions

There are several mathematical reductions that can be applied to a set partitioning problem, as described in [40, 81]. The systems of Müller [81] and Guerinik and Caneghem [48] apply reductions at a pre-processing stage of their constraint programming systems. To help us see how to apply these reductions during the search, let us envisage the backtracking algorithm as reducing the problem size whenever a variable is assigned a value; it removes at least one variable from the set of unassigned variables. Note, this may only be a temporary assignment as backtracking can occur. The smaller problem is again a set partitioning one, with a reduced set of variables and values. Hence in theory the reductions can be re-applied. Not all the reductions have the potential to benefit from more than a single application, this is why the deletion of duplicates described below is left as a pre-processor. The reductions are:

- All duplicate shifts (shifts covering an identical set of pieces of work) are removed at the generation stage. If $S_j = S_q$ for any pair $(j, q) \in J$ delete S_j.
- The subset constraint: if $D_i \subseteq D_k$ for any pair of pieces $(i, k \in I)$, $i \neq k$, then

$\forall j \notin D_i$ and $j \in D_k$, delete j from D_k. If piece i is covered only by shifts that also cover another piece k, then piece k cannot be covered by a shift that does not cover i. This has been implemented in the following way: between all variables with a shift in common there is a constraint that checks if that domain is a subset of another whenever the domain of one of the variables changes. If domain i is a subset of domain k, the constraint will remove from k's domain all the shifts that do not cover piece i. The set partitioning constraint then ensures that variables i and k have the same value.

- The one-diff constraint: this states that if only one shift that covers piece i does not cover piece k and vice versa we can make a reduction. If $|D_i - (D_i \cap D_k)| = |D_k - (D_i \cap D_k)| = 1$ for any pair of $(i,k) \in I, i \neq k$, then let shift $j = D_i - (D_i \cap D_k)$ and shift $q = D_k - (D_i \cap D_k)$

  1. If $S_j \cap S_q = \emptyset$ then shifts j and q are merged into a single shift having a cost $c_j + c_q$. (In our case, since the cost of each shift is 1, the cost of the merged shift would be 2.) Delete piece k.
  2. If $S_j \cap S_q \neq \emptyset$, then delete shifts j and q. Delete piece k.

  We can see in Figure 5.1 a Venn diagram representing this case, where each oval represents the set of shifts that cover a piece and we can see that there is only one shift in the non-overlapping part of each oval. The two shifts that cover one piece but not the other are j and q. If j and q cover no common piece then we join them to form a single shift with a cost equal to the sum of both of the shifts. This is because if shift j is picked then shift q needs to chosen so that both pieces are covered. If the shifts have a piece in common then one cannot be picked, therefore neither can be picked and this means that i and k must be covered by the same shift. This can be implemented in Solver by a constraint that checks this case whenever variables with a shift in common have their domains changed, and removes shifts if needed. There is no need to merge the shifts as the propagation of the set partitioning constraints will force the use of both if one is used. Similarly there is no need to delete the piece k as this will be assigned a value in accordance to the constraints.

To the knowledge of the author, applying these set partitioning constraints dynamically is new.

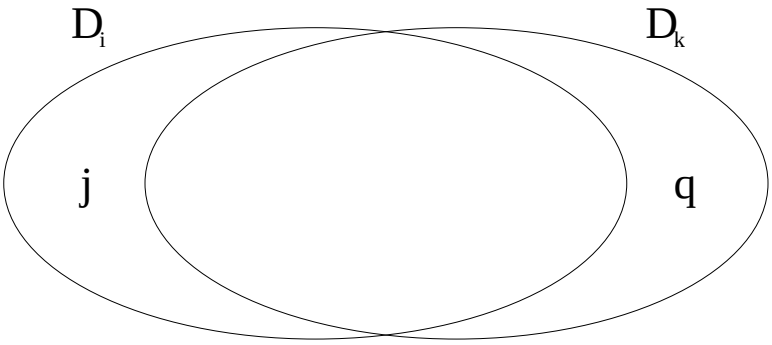

Figure 5.1: A Venn diagram of the domains of two piece variables, i and k

Table 5.2 shows the result of using the dynamic reductions. The subset constraint in general reduces the number of fails and we can see in the **r1a** problem it produces a schedule with one less shift than without the reduction. Interestingly on the **c1** data the number of shifts in the best solution found has increased by one. This we believe to be due to the effect of the smallest domain ordering and restarting from the existing solution after each new bound is placed on the problem. The subset constraint reduces the size of domains and so this will affect the ordering if we use smallest domain ordering. If we do not use these two techniques, using the subset constraint reduces the number of fails every time, but the **c1** problem is still unsolvable without the smallest domain ordering. Applying the subset reduction throughout the search is an expensive process (in terms of memory) and the decrease in the number of fails is offset by this increase. In fact using the subset constraint with the **c1a** problem the machine runs out of memory (shown by 'om' in Table 5.2) before it can find a solution. So depending on the user's needs in limiting memory or time of execution the constraint may or not be of use. This led to implementing a new way of expressing the constraint. The original constraint is posted on a pair of variables that have a value in common. Every time the domain of either of these changes, the constraint checks to see if the one with the smaller domain is a subset of the other. As it does this it stores the values that are unique to the larger domain. If the smaller domain is found to be a subset then these stored values are removed from the

Instance	t1	r1	r1a	r2	t2	r3	c1	c1a	r4
pieces	24	53	53	54	125	160	186	186	203
initial shifts	77	2503	4273	3001	3015	19091	3829	7543	2484
size	271	12k	21k	13k	17k	214k	27k	53k	17k
opt # shifts	7	11	11	14	19	16	26	26	25
rs sd:									
best result	7	18	18	19	22	om	30	31	31
fails	2	223	1437	1796	4838	n/a	921	96	9558
time (secs)	0.052	38	361	301	725	n/a	94	65	810
fails to prove opt	119	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
time to prove opt	0.45	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
rs sd Subset:									
best result	7	18	17	19	22	om	31	om	31
fails	1	123	2893	825	1004	om	397	om	755
time (secs)	0.07	39	873	199	483	om	174	om	159
fails to prove opt	162	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
time to prove opt	0.41	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
rs sd new Subset:									
best result	7	18	17	19	22	om	31	33	31
fails	1	123	2893	825	1004	om	397	7	755
time (secs)	0.11	61	1101	261	104	om	351	182	327
fails to prove opt	162	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
time to prove opt	0.96	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
rs sd One-Diff:									
best result	7	18	18	19	22	om	30	31	31
fails	2	221	1433	1788	4826	om	1033	95	9302
time (secs)	0.07	40	366	308	812	om	113	79	1006
fails to prove opt	166	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
time to prove opt	0.58	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 5.2: Results of using the reductions dynamically
sd = smallest domain ordering, rs = restart from existing solution, rn = restart from scratch, opt = optimal, om = machine ran out of memory

larger domain. However, this storage increases the memory needed. A simpler method, rather than storing values, is to dynamically post a constraint that the two variables must be assigned the same value once one variable's domain is found to be a subset of the other. Solver allows this dynamic posting of constraints and if the algorithm backtracks to the choice point before the constraint was applied, the constraint will be removed. The results of this implementation are also shown in Table 5.2. The memory used is reduced and a solution with 33 shifts is found for the **c1a** problem. However, the program runs out of memory after this solution is found and so produces a worse result than without using the reduction. Further, this implementation tends to take more time as Solver has to re-check what values are in the larger domain that are not in the smaller domain and then has to remove them.

The one-diff constraint on most of the problems makes little impact on reducing the number of fails as the situation where the reduction can be made does not occur often.

These reductions are not used in the final system due to the fact that an efficient implementation for the subset constraint has not been found and the one-diff constraint has little practical use. However, the subset constraint does generally reduce the number of fails to find a solution and sometimes dramatically. If an efficient implementation could be found it could prove to be a useful constraint for solving set partitioning problems.

5.5 The extended model

In ILP, branch-and-bound can be used to find a good or optimal integer solution from the LP optimum. This section will further explain the branching strategy used in TRACS II as was described in Section 4.3.2.6 and how it is adapted to be used in this constraint programming system. The standard approach is to choose a variable (in driver scheduling, a shift) whose value in the LP optimum is fractional (in this case, strictly between 0 and 1) and to form two branches: on one branch, this variable is forced to have the value 0 and on the other, the value 1. A new optimum solution is formed in each case, followed

by the formation of further branches, and so on. A branch terminates if either an integer solution is found, or its value is greater than the best integer solution already known. In the development of IMPACS, it was found at an early stage that this form of branching (*variable branching*) is completely ineffective for driver scheduling problems, for reasons similar to those given in Section 5.2.1. The alternative branching strategy developed for IMPACS and later used in TRACS II is relief time branching. This assigns a (possibly) fractional value to each relief time (strictly, relief opportunity) in the bus schedule, based on the current non-integer LP solution. This is the sum of the values of the variables representing shifts which finish a spell at that RO. A branch is then formed by choosing an RO for which this value is fractional (again, strictly between 0 and 1). The value is forced to be 0 on one branch (which means that all shifts starting or finishing a spell at that RO are banned) and 1 on the other (which means that all shifts covering both pieces of work immediately before and after this RO are banned). This branching strategy was found to be very successful, and incomparably more useful than variable branching. Choosing which ROs are to be used does not explicitly choose the shifts to use. However, once the ROs have been set the choice of shifts is reduced dramatically and the problem becomes trivial.

This experience prompted us to implement RO branching in our constraint programming process. We have a set of variables $\mathcal{R} = \{R_k, k = 1 \ldots r\}$ where R_k is an "active" RO and r is the number of such ROs. Active ROs are the ones that we need to choose a value for, i.e. ROs which start or end a bus are excluded, as these have to be used. So $r = m - b$ where $b =$ number of buses and as before m is the number of pieces of work. These variables have a binary domain with values 1 (use) or 0 (do not use). For each RO variable R_k there is a corresponding piece variable P_i such that the RO with index k is the start of the piece of work with index i. We then set up a constraint

$$R_k = 1 \longrightarrow P_{i-1} \neq P_i \tag{5.2}$$

$$R_k = 0 \longrightarrow P_{i-1} = P_i \tag{5.3}$$

for: $k = 1, \ldots, r; i = 2, \ldots, m$.

So if an RO is used, its adjacent pieces must have different values i.e. be covered by different shifts and if it is not used, its adjacent pieces must have the same value. This can be seen as an extension to the second model, the pieces as variables model. The complexity of the RO extended model is 2^r (where r is the number of ROs) which is less than the previous two models and it avoids the "all or nothing" choice of shifts. As stated above, once the choice of the RO variables is made assignments to the piece variables is in practice trivial. In effect by using ROs as variables we are making higher level decisions than what shifts to use. The effect of these decisions then propagates to the piece variables and so the choice of shifts. If we make the right higher level decisions we have to make fewer decisions than if we just used low level decisions to get a solution.

We order the RO variables first, followed by any piece variables that have not already been assigned a value. We have investigated several orderings for the set of RO variables:

1. Ordering by adjacency, where we order starting from the first RO on the first bus then the second on the first bus, etc. until the last on the last bus. So we are dealing with the ROs on a bus in order of their time. However, the ordering of the buses is generally arbitrary.

2. Ordering by choosing first the ROs that cut out the greatest number of shifts.

3. Ordering by time of day, where we pick the variables in order of time of day, earliest first. This is similar to the way some human schedulers build up a schedule.

At this stage the adjacency ordering produced the best results with the least number of fails.

The first attempt at value ordering involved a greedy process of binding an RO variable to

Instance	t1	r1	r1a	r2	t2	r3	c1	c1a	r4
pieces	24	53	53	54	125	160	186	186	203
initial shifts	77	2503	4273	3001	3015	19091	3829	7543	2484
size	271	12k	21k	13k	17k	214k	27k	53k	17k
opt # shifts	7	11	11	14	19	16	26	26	25
basic RO:									
best result	7	12	12	14	n/a	om	n/a	n/a	n/a
fails	721	402	29	2562	>10k	n/a	>10k	>10k	>10k
time (secs)	1.60	44	31	289	>962	n/a	>1k	>3k	>567
fails to prove opt	>10k	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
time to prove opt	27	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 5.3: Results of using the RO with greedy ordering and adjacency

0 as first choice (i.e. not using the RO). So as the program goes through the RO variables it chooses not to use each RO until a fail occurs and then it sets the current RO variable to be used. The principle behind this heuristic is that we will tend to use fewer shifts if we use fewer ROs. It also tends to maximise spell length which is similar to the way a human scheduler goes about the task, although a human scheduler would use informal heuristics and intuition to decide when to use a shorter spell length. By using the extended model with the adjacent ordering and this greedy value ordering for the RO variables, there was a general improvement in performance. Table 5.3 shows that an optimal solution was found for the **r2** problem for the first time. However the program could not prove this was optimal. For the larger problems no solution can be found. This is because combinations of assignments are made by the greedy heuristic early in the search process that cannot lead to a solution. The resulting fail only occurs later in the search process and the algorithm never backtracks far enough to undo the early errors.

5.6 Using The Relaxed LP Solution

When TRACS II forms schedules it first solves the relaxed LP problem for the generated set of shifts. The relaxed LP problem is the set covering problem without integrality constraints on the shift variables. The method used to solve this problem is detailed in Section 4.3.2.5 and [37].

The relaxed LP solution is an assignment of possibly fractional values to shifts, in which the sum of the shifts covering any piece of work is greater than or equal to 1. The number of shifts used in this solution, i.e. the sum of the possibly fractional values, gives us a lower bound on the optimal number of shifts. In practice rounding up the number of shifts to the next higher integer almost always gives the optimal number of shifts.

Although the relaxed LP solution is not a feasible driver schedule, we can plot it as if it were. Figure 5.2 shows the coverage of a running board in such a solution. Each fractional value of a spell is the sum of all the fractional values of the shifts containing that spell. We can see in this example, that the sum of the fractional values of the spells covering each piece of work is 1, so there is always, mathematically, exactly one full driver (made up of fractional drivers).

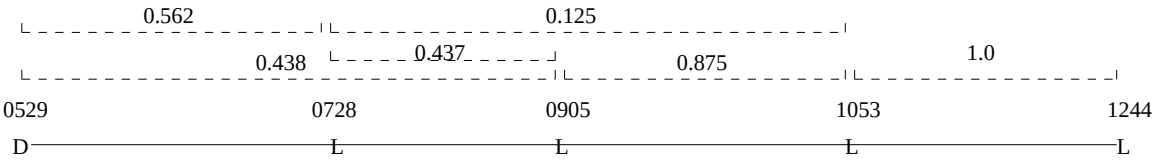

Figure 5.2: Fractional coverage of a running board

We investigated the fractional solutions of several problem instances in search of common features that we could take advantage of. Some ROs have fractional shifts starting then. Out of these a high proportion had shifts starting then in TRACS II's final schedule. This observation led us to the first attempt to use the LP solution as a guide, although we will describe later why it was unsuccessful and was replaced by the second attempt. We first try to solve the subproblem of choosing which ROs will have shifts starting at them. Once this subproblem is solved then the rest of the problem will be trivial. We can use the LP solution by guiding the choice of which ROs to use as starting ROs. So as a heuristic for choosing the ROs that will be starting ROs in the final solution we can choose all the ROs with shifts starting then in the LP solution.

To integrate this into our program we adapt the RO variables to have triples for their domain: use as a start (0), use but not as a start, (1) and do not use (2). If we use the

Instance	t1	r1	r1a	r2	t2	r3	c1	c1a	r4
pieces	24	53	53	54	125	160	186	186	203
initial shifts	77	2503	4273	3001	3015	19091	3829	7543	2484
size	271	12k	21k	13k	17k	214k	27k	53k	17k
opt # shifts	7	11	11	14	19	16	26	26	25
nostart:									
best result	n/a	n/a	16	15	n/a	om	n/a	n/a	n/a
fails	>10k	>10k	6819	30	>10k	n/a	>10k	>10k	>10k
time (secs)	>4	>1.6k	1168	14	>972	n/a	>2k	>1.4k	>510
fails to prove opt	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
time to prove opt	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 5.4: Results of using the RO model with domains of triples
rs = restart from existing solution, nostart = using triple RO domain, om = machine ran out of memory

RO as a starting RO (0) we not only remove all the shifts that do not have a spell starting or finishing at the RO but also all the shifts that do not have their *first* spell starting at it. If we use the RO but not as a starting RO (1) then we remove shifts that do not start or end a spell then, as well as shifts that have their first spell starting then. If we do not use the RO (2) then we remove all shifts that have spells starting or ending then. There is a constraint that imposes the implications of an RO variable's (R_k) assignment on the piece variables. It is associated with the RO in question and the two piece variables that correspond to the pieces of work on either side of the RO (P_{i-1} and P_i). This is because the constraint may have to remove values from both of the piece variables' domains, as shifts with spells starting at the RO will be in the domain of P_i but not in P_{i-1} and vice versa for shifts with spells finishing at the RO.

The search firsts assigns values to the RO variables that were starting ROs in the fractional solution. These are given the value 0. The rest are assigned value 2. The variables were ordered according to adjacency as described in the previous section.

The results for this heuristic are shown in Table 5.4. The results obtained doing it this way are worse than not using the guide and just using the greedy heuristic. The reason for this we believe is that we are increasing the size of the problem greatly by having three choices instead of two for each RO. It was hoped by concentrating on the starting ROs we

would in practice be decreasing the effective size of the problem. However, for this to be true either a large amount of propagation or a very good value guide is needed. It seems that there is not enough propagation and the value guide is not good enough in this case.

In the TRACS II system fractional values of ROs are used to guide the branch and bound process. Fractional values are assigned to an RO by adding up the fractional values of all the shifts starting (or all finishing) at it. So, for example, if we return to Figure 5.2 we see that the RO at time 0728 has a fractional value of 0.562, as the sum of the fractional spells starting at that time is 0.125 + 0.437. Smith [99] implemented a heuristic reduction that before going into the branch and bound process removed all ROs that in the relaxed LP solution had zero value and were not used by shifts in the basic feasible solution[1]. This reduction greatly decreases problem size and has little - if any - detrimental effect on the quality of the final solution. We have adopted this approach and remove all zero value ROs.

5.6.1 Value and variable ordering

The fractional values are further incorporated into our process as a guide to value choice. If the fractional value is greater than 0.5, then the first choice of a value for the RO variable is 1; otherwise it is 0.

We also use the fractional values by ordering the variables according to closeness to integrality i.e. closeness to 1 or 0. So we begin by choosing to use ROs whose value in the LP optimum is 1 or close to 1, and not to use ROs whose value is 0 or close to 0. In terms of the bus schedule, this is a slightly unnatural way of processing the RO variables as the algorithm will jump around the different buses, maybe only setting a value for one RO before jumping to the next bus. However, we think that it is sensible that we should set the values of variables that we are most sure about first. This assumes that fractional values that are closest to 1 are most likely to be used and those closest to 0 are least likely

[1] Occasionally, a shift in the basis at zero value uses an RO which has itself zero value. Because TRACS II uses the LP optimum basis as the starting point of the search for an integer solution, it may need to keep such a shift.

to be used.

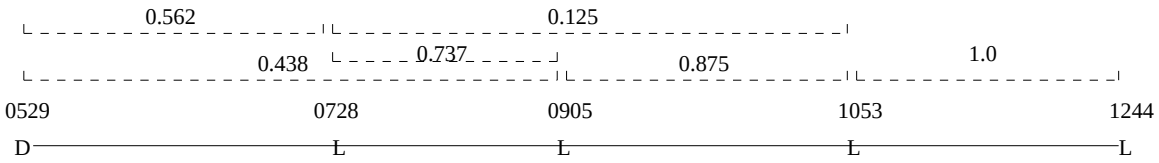

Figure 5.3: Fractional coverage of a running board with over-cover

The LP solution is set covering and so may contain over-cover, so the fractional value of an RO could be greater than one. Further, an RO could have two different values depending on whether we summed the shifts starting or finishing at it. We can see this in Figure 5.3. If we sum at the start then the RO at time 0905 has a fractional value 0.875 but if we sum the end of shift we get a value of 1.175. This fractional over-cover could cause problems, as we are trying to use the fractional values to guide us to a set partitioning solution, but the fractional values could correspond to a relaxed set covering solution and so be a poor guide. In practice none of the problem instances in the results have over-cover in their LP solution. However, we have found it in a problem instance that we cannot find a solution to. It is unlikely that this over-cover is what is stopping us from find a solution because the problem instance in question is much larger than the test problems we have shown results for[2] and so it would be unlikely that we could find a solution whether there was over-cover or not. So at the moment we do not need to consider this situation further than to propose a way of tackling it. This would be done by leaving any ambiguous fractional values till last, by then propagation will probably have set the value of the RO(s) in question anyway.

5.6.2 Additional constraints and heuristics to improve efficiency

In the relaxed LP solutions we noticed that in several cases the sum of the fractional values of pairs of adjacent ROs on the same bus was 1 (for example in the **r1** there are two adjacent ROs with values 0.24 and 0.76). An observed characteristic of such pairs

[2]The problem has a larger number of pieces, 242, and shifts in the optimal shifts, 29, than the test problems. It has 2202 shifts

was that in the integer optimal solution only one of the pair would be used. Therefore, we added a constraint between such variables, stating that only one of them could be used. With the success of this heuristic we expanded it to triples of variables whose fractional values summed to very close 1. We call these the Combo constraints; the double Combo for the pairs and the triple Combo when we have three adjacent ROs (for example with values: 0.12, 0.24 and 0.64).

An additional way of aiding the search was found by examining the structure of the bus schedule. From this it is clear that in *theory* the RO model is open to extra propagation on the values of ROs. For example, if we use RO A we cannot use the following RO B if no spell starts at A and ends at B. So when there is no such spell we can set the variable corresponding to B not to be used if we choose to use A. This propagation would not be normally inferred by the current constraints, unless the domains of the pieces of work adjacent to B became identical. We have implemented two ways of dealing with this situation: the first is to deal with it in pre-processing and the second is to deal with it during the search.

In pre-processing we set up constraints between adjacent ROs that do not have spells between them, stating that if one is on the other is off. We have also implemented this constraint so it can act during the search. This is because shifts are removed during the search, therefore this situation may occur during the search. We have therefore implemented a constraint that watches for this situation during the search. Once found it is dealt with in the same way as the pre-processing constraint. This takes more time than the pre-processing constraint, as we have to check each time a constrained RO gets a value. However, both extra propagations on the ROs (dynamic and pre-processing) in practice have no impact on the solution or how many fails it takes to be obtained. In two out of the 9 test cases they both removed two choice points but made no difference in the other test cases. The reason for this is probably that the LP solution value guide implicitly caters for this situation.

5.6.3 Related work

Related work has previously been described in Section 4.4. In this section we will relate the systems described there that use mathematical programming combined with constraint programming with the work shown here.

The previous constraint programming systems that have used the relaxed LP solution have used it in a different way to ours and have not used the structure of the problem to increase the usefulness of the solution. Guerinik and Caneghem [48] and Rodosek *et al* [87] use the fractional value of a shift (rotation in air crew scheduling) as the guide to the first value chosen for their shift variables. They take slightly different approaches in their search. In Guerinik's and Caneghem's paper the variables are ordered according to their closeness to 1, the closest first. The value first attempted for each variable is 1. So while there are no fails the values of the relaxed LP solution are consistent with the choices made. However, when a fail occurs a variable will attempt the value 0 and by so doing the relaxed LP solution will no longer be in accordance with the partial constraint programming solution and therefore the relaxed problem will be re-solved. Rodosek *et al* order the variables according to their closeness to 0 or 1 (closest first) and choose the nearest integer value to the fractional value as the first choice. This is more like the way that we use the LP solution than the method by Guerinik and Caneghem. However, Rodosek *et al* resolve the relaxed problem whenever there is a fail. In this way the fractional values are affected by previous decisions and so become a more accurate prediction of what the final integer values will be.

In our system we do not re-solve the relaxed LP, thereby making our process less dependent on the LP techniques and so maintaining the flexibility of the constraint satisfaction formulation. We can envisage a scenario in which we might solve the basic relaxed LP and then add any constraints that are hard to express in the LP formulation, finding an integer solution using constraint programming. We are currently investigating situations where such constraints may occur. It is worth noting that by adding these constraints, the LP solution will become less applicable to the final solution, which is why it is only

used as a guide.

The major difference between our use of the relaxed LP solution and the use in the two systems described above is that they use the fractional value of a shift and we use the fractional value of an RO. This is similar to the difference between variable branching and RO branching in IMPACS as discussed in section 5.5. The fractional value of a shift is likely to be of less use than the fractional value of an RO. Several shifts may cover similar sets of pieces of work and so if a shift has a high fractional value then it is likely that a similar shift will be used but not necessarily this particular shift, whereas if an RO has a high fractional value it is likely to be used.

5.7 Results

Using the final version of the program we have obtained the optimal number of shifts in all problem instances. Without using the RO variables an optimal solution could only be found for the very small **t1** problem. A summary of results can be seen in Table 5.5. We have shown the results of all the heuristics that were tested in the final development stage of the system. For each of these we have the number of fails to produce an optimal solution. In all cases we use RO variables and fractional values of these as a value ordering guide.

The double Combo constraint makes a significant reduction in the number of fails in several problem instances. Moreover there is only one case where it has a detrimental effect, which is when using adjacency ordering (Section 5.5) on **r1a**; it did not find an optimal solution after 50000 fails. Yet this does not matter, because we use closest to integer ordering (Section 5.6.1) in the final system since in the test cases it always produces the optimal in no more fails than the adjacency ordering. This fact also makes the triple Combo constraints obsolete and so they are not incorporated in the Table 5.5 because that constraint only makes a difference for the adjacency ordering but not for closest to integer ordering.

The best set of heuristics is to use the extended RO model, using the relaxed LP solution as a value and variable ordering guide. The extended model has reduced the complexity of the problem and allowed us to make better use of the relaxed LP solution. So the formulation of the problem makes an enormous difference, not only in reducing the complexity of the problem but also in enabling better search strategies to be used. The most successful variable ordering is the closest to integer ordering. The double Combo constraint is a useful constraint and is incorporated into the final system.

The last row of the table shows how a new implementation of the *rem* constraint (Section 5.2.2) speeds up the constraint handling process and so speeds up the algorithm. In the first implementation a constraint was set up between a pair of variables that had a value (shift) that was common to both domains. Every time the domain of one of the variables changes the constraint checks to see if the shift associated with it has been removed from one of the domains that has changed. If this is the case then that shift is removed from the other domain. The new implementation only has one constraint per pair of variables that have a shift common to both of their domains. The constraint makes use of the fact that at every choice point Solver stores the values removed from each domain. If there are any values removed from a domain in a pair of constrained variables, the constraint cycles through the store of these that Solver retains and removes them from the other domain in the pair.

With problems tested that were larger than the ones shown the final algorithm could not find a schedule with the optimal number of shifts in the allowed number of fails. However, size is hard to measure as the number of potential shifts, the number of pieces and the number of shifts in the optimal schedule all affect the size. We have defined size as the size of the CSP (see Section 2.7). Despite this, it is a mistake to directly relate this measure to how hard a problem is to solve. We can measure how hard a problem is to solve by running an algorithm on it and seeing if the algorithm can solve it. If it can solve it we measure how long it takes to solve it and use this as a measure of difficulty. However, there is no algorithm independent measure of hardness and this remains an open question for the driver scheduling problems and for CSPs in general. We believe failures to find

Instance	t1	r1	r1a	r2	t2	r3	c1	c1a	r4
pieces	24	53	53	54	125	160	186	186	203
initial shifts	77	2503	4273	3001	3015	19091	3829	7543	2484
size	271	12k	21k	13k	17k	214k	27k	53k	17k
opt # shifts	7	11	11	14	19	16	26	26	25
fails	49	10	3713	22	>50k	>50k	877	5113	28822
time (secs)	0.87	1.05	172	2.00	>1.8k	>12k	10.90	56	229
c:									
fails	1	10	>50k	22	>50k	>50k	3	2	1519
time (secs)	0.04	1.05	>2k	1.97	>1.5k	>12k	1.05	1.22	13.21
po:									
fails	45	121	303	22	3118	5942	68	191	496
time (secs)	0.08	2.73	17.97	2.05	94	1525	72	3.79	7.70
po c:									
fails	0	10	228	22	174	5942	1	1	9
time (secs)	0.06	1.03	15.44	2.04	7.81	1525	1.04	1.22	1.26
Final system									
po c nr:									
fails	0	10	228	22	174	5942	1	1	9
time (secs)	0.03	0.08	13.42	1.65	6.12	1078	0.95	1.11	1.17

Table 5.5: Final results for constraint programming system
nr = new implementation of the *rem* constraint
po = closest to an integer value ordering
c = using the double combo constraints,
opt = optimal

solutions for larger problems may be due to the systematic backtracking search system. To illustrate this, let us say there is an RO that has a value close to 1 in the relaxed LP solution, our program would set this to be used. It may then make many more decisions and, due to the size of the problem, never be able to backtrack to change that decision. So if it is crucial not to use that RO the program will never find an optimal solution.

5.8 Flexibility of CP model

Some areas where the ILP technique has been found lacking are discussed in Section 4.3.2.8. An advantage of CP over ILP is that the CP approach is more flexible in its expressiveness. This flexibility was originally one of the reasons why the CP method was tried. A

possible area when an advantage might be found is with windows of relief opportunity (see Section 4.3.2.8).

Windows of relief opportunity would be difficult to represent in any set partitioning/covering formulation, as such formulations deal with specific hand-over times. However, constraint satisfaction may provide the key. The constraint programming approach builds up a schedule, and it may be possible to create some shifts during the process, in particular when a fail occurs. If the fail occurs due to an assignment of a relief opportunity variable, it might be possible to adjust the time of the relief opportunity and generate new shifts. Much research would be needed to develop and test this idea. Alternatively, using one RO for every minute in the window may not cause the same problems for CP as it does for ILP. In CP a constraint could be set up to specify that only one RO within the time window could be used. This local constraint cuts the effective size of the problem, unlike adding a similar constraint in an ILP model.

5.9 Conclusions

We have used both a pure constraint programming approach and an improved hybrid CP/LP approach for solving real world problems of driver scheduling. The program's limited use of the relaxed LP solution brings an amount of independence that will allow the flexibility of the CP approach to be taken advantage of fully. The model, constraints, and variable and value ordering, have been specially developed to take advantage of the constraint programming formulation and the driver scheduling problem structure. The domain specific knowledge incorporated allows us to solve set partitioning problems of sizes beyond the reach of pure constraint programming systems.

ILP based systems such as TRACS II are still faster and can produce solutions for much larger problems than this system. Nevertheless, it is hoped that the advantages and flexibility of constraint programming will be useful in adding further constraints that are hard to model in an LP formulation as discussed in Section 5.8.

Although we have described the constraint satisfaction system we have developed in terms of shifts and pieces of work, all the models, reductions and search methods before Section 5.5 could be applied to any set partitioning problem. The pieces of work would then correspond to the elements of the set I and the shifts to the set of subsets S, as referred to in Section 5.1.

This research has been very domain specific. However, it has highlighted several considerations that are useful for modelling practical constraint satisfaction problems. These are the following:

1. Variable ordering in practical problems (Sections 5.3 and 5.6.1). We have seen that the smallest domain dynamic ordering is not necessarily better than ordering based on the structure of the problem. The ordering based on the structure works well because it is not purely random, it groups the pieces of work according to the bus they are on and what time of day it is. The advantages of this are discussed in Section 7.3. The conclusion is that in practical problems a natural ordering may occur that takes advantage of hidden structure in the CSP and is therefore better than a general ordering heuristic.

2. Adding heuristic constraints, that may remove solutions from the search space. An example of this is the Combo constraint 5.6.2. These heuristic constraints will be useful in hard to solve structured problems where there is already no guarantee of finding a solution in the required operational time of the company. There have been no rigorous studies of this type of constraint even though as we have seen in this work they can be more useful than adding implied constraints.

3. Mathematical reductions during the search (Section 5.4). It may be possible to carry out mathematical preprocessing steps during the search. The important thing to do when adding this type of implied constraint is to weigh up the extra constraint propagation processing that has to be done against the reduction in the number of fails. We have seen also that how the constraint is implemented can make a large difference to performance (both in memory and in time).

4. Higher level decisions. We can see that using the extended model with ROs as variables allows high level decisions to be made, rather than just choosing shifts. However, without a good value guide these high level decisions increase the size of the problem as there are more decision variables. In a problem where constraint propagation did more pruning, incorrect value assignment might be detected early, but in this problem it is essential to have a good value guide. When the LP solution is used as a value guide the solutions improve greatly. This can be taken on board by developers working on other practical problems. Taking higher level decisions before low level decisions can make a great difference to solution quality. Higher level decisions are related to domain splitting (at each branch of the search removing a portion of the domain). Although domain splitting causes less propagation than assigning a value to a variable, if a good branching heuristic (value guide) can be found then it can be more effective.

It is clear that there is further work that could be done on the implementation of the constraints in the model. A more efficient implementation of the subset constraint could reduce the time needed to produce a solution without undue increase in the use of memory. The improved version of the *rem* constraint has speeded up the algorithm. It would also be of use to investigate new ways of expressing the constraint that sets variables that have a value in common to that value if one of them is set to it. This has so far resisted attempts to improve on its representation.

Chapter 6

Using GENET on the Driver scheduling problem modelled as a Constraint Satisfaction problem

6.1 Introduction

Much of the work in this Chapter has previously been published by Curtis *et al* [20].

Local search methods have hitherto not had much success in the construction of bus driver schedules. An exception is the application of genetic algorithms to bus driver scheduling described by Kwan *et al* in [67] and Section 4.5.2. Given a solution of reasonable quality, it is often possible to make minor adjustments to individual shifts, and still maintain the legality of the schedule: this is for instance how we eliminate over-cover if there is any in the best solution found by TRACS II. However, it is very difficult to make major

improvements, for instance on the scale required to reduce the number of shifts in the solution, unless the existing solution contains gross over-cover. If there is little over-cover or none at all, the changes required to eliminate a shift would entail simultaneous changes to many other shifts in the solution, which would be difficult for a local search procedure to find. Furthermore, investigations by Kwan [65] have suggested that, for some problems at least, the number of possible schedules with the minimum number of shifts is very small. When there are very few solutions, or in this case very few optimal solutions, local search is expected to perform poorly.

However, GENET is a local search procedure which has been successfully applied to constraint satisfaction problems of several kinds, including optimisation problems (Section 3.6.5). For this reason it was deemed worthwhile to investigate whether it would give good results on the driver scheduling problem. Although its performance is not comparable with TRACS II, we have been able to achieve considerable improvements over the initial simplistic model. We believe that the experiences shown here would be useful to others using GENET to solve large difficult constraint satisfaction problems and in particular problems with similar optimisation criteria.

In Section 5.1 we gave reasons for using a set partitioning formulation. However, using this formulation restricts the range of problems we can solve as we can only solve problems with a set partitioning solution. The ideal would be to have a formulation that has the guiding nature of set partitioning but the flexibility of set covering. With GENET we can achieve this as GENET only tries to minimise the number of constraint violations and is not restricted to solutions which satisfy all the constraints. Effectively, we can work with a set partitioning formulation but accept set covering solutions.

In the systematic approach detailed in the previous chapter the basic formulation of the problem is to have the pieces of work as the variables, the *workpiece model*. The chapter also mentioned another possible way of representing the problem would be to do it in the same way as the mathematical programming approach. In this *shift model*, the shifts are the variables. The domains are binary, with values representing whether to use the shift

or not use it.

The workpiece representation has two main advantages over the shift model. As discussed in Section 5.2.2 in the latter model, the number of decision variables is much larger. Furthermore, the number of assignments of values to variables in the shift model is 2^n, where n is the number of shifts. This is much larger than the number of possible assignments in the workpiece model, which is $\prod_{i=1}^{m} n_i$ (where n_i is the number of shifts covering piece i). Although with constraint propagation not all the possible assignments will be tried, the number of possible assignments gives an indication of the complexity of the model. In the systematic approach this gave a large advantage to the workpiece model. However, with a local search method the possible number of assignments (size) is less important to the algorithms ability to find a near optimal solution. This is because the local search method will only try a fraction of the possible choices, whereas the systematic approach will implicitly try all possibilities. The actual importance of size is problem specific and is dependent on the type of systematic and local search methods used.

Another disadvantage of the shift model is that there must be constraints in place to ensure that no pieces of work are left uncovered. If those constraints are violated, the solution is not a feasible schedule and cannot easily be converted to one, unlike a solution with over-cover. On the other hand, when using local search with the workpiece model, every state of the search could be a schedule (however inefficient). This is the reason we have opted to maintain the workpiece model in the GENET system.

One of the drawbacks of a stochastic method such as GENET is the loss of a guarantee of producing an optimal solution. Given time, an exhaustive search will always find an optimal solution whereas a stochastic method may not. However, the fact that we have a heuristically reduced set of shifts means that we have sacrificed the guarantee of a real optimal solution and aim to produce near optimal or possibly optimal solutions. So the guarantee is already lost. Moreover, if a local search method can deal with a large initial set of shifts then fewer heuristic reductions need to be done and there is less chance of removing useful shifts.

6.2 The GENET model

The constraints in the set partitioning formulation are binary and so could be set up in the original format of GENET (see Section 3.6). If any two variables (pieces of work) have a shift that covers both of them then we need to set up constraints to deter one variable from choosing that shift and the other one not. Hence, the network could be set up as in Figure 6.1. The cluster of three neurons on the left represents variable A and the cluster of two on the right represents variable B. The value associated with each neuron is the number of the corresponding shift. The symmetrical weighted connections are shown by lines between the neurons: the weights are initialised to -1. The pieces of work could both be covered by shift 5 and so if one chooses 5 and the other does not, over-cover will occur. We want to deter this from happening by having *nogood* connections between label nodes. To illustrate how the network would work let the labels `<A,3>` and `<B,5>` be *on*. Label `<A,3>` and `<B,5>` therefore output a 1; labels `<A,1>` and `<A,3>` receive an input of -1 from `<B,5>` and label `<B,5>` receives an input of -1 from `<A,3>`. Other inputs remain at 0. When we repair variable A the label, `<A,5>` is turned *on* as this has the largest input (0). So now there is no conflict between variables A and B; the same shift is covering both.

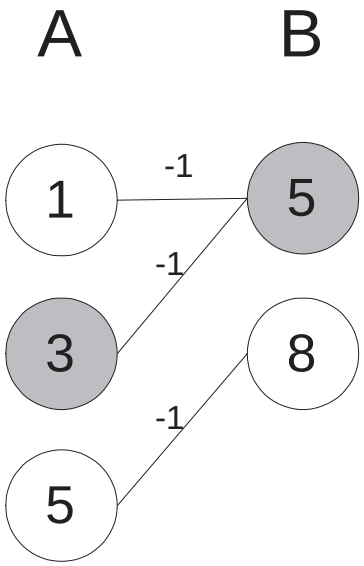

Figure 6.1: Two node clusters with set partitioning constraints in GENET

When we use pieces as variables and shifts as values and allow constraints to be broken we introduce a mapping of one to many from the set of possible solutions to the set of possible states of GENET. This means that a variable might change values and yet the schedule would remain the same. If the variables were the shifts and the values were 0 and 1 then this would not happen. However, in a local search method this is not as much of a hindrance as this type of symmetry would be to an approach that searched exhaustively. Instead of stepping from a schedule to a different schedule, having this symmetry can be viewed as allowing sub-steps that will eventually lead to another schedule.

Unfortunately, connecting all pairs of inconsistent labels takes up too much memory. Only the smallest problem (**t1**) in the test data can be represented using GENET's existing binary-constraint representation. To combat this a new constraint neuron has been developed in a similar way to GENET's non-binary constraints described in Section 3.6.3. Figure 6.2 illustrates the use of the new neuron to represent the constraint between the pieces A and B.

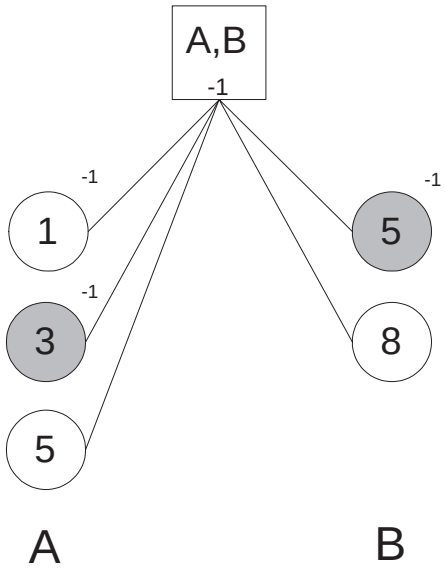

Figure 6.2: Set partitioning constraint node in GENET

The connections are no longer symmetric; if a label node is *on* then it outputs a 1 to the constraint neuron. The constraint neuron then decides, knowing which label nodes it has received an input from, which label nodes need to be penalised. It then sends an output to

those nodes which will be negatively weighted by the constraint node's stored weight. As before labels `<A,1>, <A,3> and <B,5>` receive an input of -1 if `<A,3> and <B,5>` are *on*. Each constraint node has only a single weight, initialised to -1, so that all label nodes that are penalised by a constraint are penalised equally. This is different to the representation described above, where the weights of individual connections between two node clusters may become different through the learning process. Hence, the energy landscape will become different for the two models.

6.3 Sideways moves

The first consideration is to investigate the suitability for this particular problem of sideways moves (see Section 3.6.2). In such a highly structured problem it is unlikely that sideways move would be useful. The choice of value that a variable takes is so dependent on the choices that other variables make, that changing these values without reason is unlikely to lead to improvement. This theory is borne out by the results in Table 6.1. The table shows the results for the 9 problem instances described in Section 5.3. The final column shows the average of these. For each problem instance the program was run 10 times with 10 different seeds for the program's random number generator. A run is terminated after 3000 cycles. This is because, although the CPU time to find the best solution is often short, the user time can be much longer and the program has to be run 90 times to test each heuristic. All the runs of the program where done on an SGI Octane machine. This is a different machine to the one used for the constraint programming approach described in Chapter 5. The reason for this is that, due to the restrictions imposed by the Solver licence and an implementation issue with GENET, each algorithm cannot be run on the machine the other was run on. This means that the timings are not directly comparable. Sample runs were made up to 10000 cycles but no extra improvements were made on the solution. Listed in the results is the average of the number of shifts in the best solution found for each run. The standard deviation is not given but for the basic model the average standard deviation is less than 1 whole shift (the average standard deviation for the

Instance	t1	r1	r1a	r2	t2	r3	c1	c1a	r4	av
pieces	24	53	53	54	125	160	186	186	203	n/a
initial shifts	77	2503	4273	3001	3015	19091	3829	7543	2484	n/a
size	271	12k	21k	13k	17k	214k	27k	53k	17k	n/a
opt # shifts	7	11	11	14	19	16	26	26	25	17.2
basic:										
av. # shifts	7.20	16.8	18.2	19.6	23.5	21.8	34.9	36.2	33.3	23.5
av. time (secs)	0.09	0.06	0.03	0.03	0.33	1.00	0.84	0.92	0.87	0.46
best # shifts	7	15	16	19	23	21	34	34	32	22.3
time (secs)	0.01	0.20	0.03	0.02	0.17	0.64	0.68	1.12	1.24	0.46
lsw:										
av. # shifts	7.70	16.7	18.5	19.5	23.8	21.7	35.2	36.2	33.6	23.7
av. time (secs)	0.11	0.16	0.08	0.09	1.83	11.7	5.3	5.02	3.78	3.12
best # shifts	7	16	18	18	23	20	34	35	32	22.6
time (secs)	0.01	0.09	0.05	0.22	0.67	8.4	7.10	3.77	3.0	2.59
av. # shifts	7.20	17.1	18.4	19.6	26.4	41.9	51.4	52.1	47.5	31.3
av. time (secs)	0.11	0.60	0.26	0.29	3.09	8.30	0.92	0.77	3.95	2.03
best # shifts	7	16	17	18	23	34	46	50	43	28.2
time (secs)	0.01	0.59	0.20	0.77	6.11	18.5	7.74	0.27	13.1	5.26

Table 6.1: Results on allowing or not allowing sideways moves.
lsw = limited sideways moves, fsw = full sideways moves

final model is also less than 1). Also shown is the average time at which these solutions were found. The final two rows give the lowest number of shifts achieved out of all 10 runs and the time it took to find this solution.

The results using limited sideways moves and no sideways moves are very similar. Although in some cases one is better than the other, neither has a significant advantage. However, the full sideways moves strategy is much worse than the other two especially for larger problems.

6.4 Superfluous/redundant shifts

An extreme situation that can occur when allowing constraints to be broken, so allowing over-cover, is that the shift that is selected by some variable might not uniquely cover any piece of work, i.e. every piece of work covered by this shift is also covered by another selected shift. In examining states of the network it was found that at times this did

shift	pieces covered
88	6, 8
86	6, 8, 14
135	8, 14
173	13
170	13, 14, 15, 16
177	15, 16

Table 6.2: Example shifts used in a state of GENET.

happen. These superfluous shifts can be removed, thus reducing the number of shifts without leaving any uncovered pieces of work. To tackle this situation a routine was devised to take action at local minima.

There may not be a unique way of removing superfluous shifts; for instance, if one piece of work is covered by two superfluous shifts and no other shift, then when either shift is removed the other is no longer superfluous. This means that shifts may potentially be superfluous but in fact may become needed if other potentially superfluous shifts are removed. We will use a real set of shifts that were in use in a state of GENET to illustrate this. Table 6.2 shows the index of each shift and the numbers of the pieces of work that it covers.

If we remove shifts 86, 170 then there are no conflicts and all the pieces are still covered. We will discuss how we might translate this process into a general formula for removing shifts.

There are several possible general strategies. For instance:

1. Remove all shifts that are a subset of another shift. In the example above we would remove 88, 135, 173, 177. This would still cause a conflict between 86 and 170 but would leave only 2 shifts being used.

2. Remove potentially superfluous shifts that are a superset of another (e.g. 86, 170). This leaves no conflicts but uses 4 shifts.

3. Solve the problem of finding an optimal set of superfluous shifts to remove as a

Instance	t1	r1	r1a	r2	t2	r3	c1	c1a	r4	av
pieces	24	53	53	54	125	160	186	186	203	n/a
initial shifts	77	2503	4273	3001	3015	19091	3829	7543	2484	n/a
size	271	12k	21k	13k	17k	214k	27k	53k	17k	n/a
opt # shifts	7	11	11	14	19	16	26	26	25	17.22
rem:										
av. # shifts	7.10	16.3	18.4	18.7	23.2	21.0	33.9	35.0	31.8	22.8
av. time (secs)	0.02	0.02	0.01	0.01	0.16	1.24	0.28	0.38	0.29	0.27
best # shifts	7	15	16	17	23	20	33	34	31	21.8
time (secs)	0.00	0.04	0.05	0.01	0.09	0.34	0.22	0.31	0.13	0.13

Table 6.3: Results of removing superfluous shifts.
rem = remove superfluous shifts

separate subproblem.

4. Repeatedly randomly remove a potentially superfluous shift until there are no superfluous shifts left.

5. Look at the overall energy change of removing each shift. Remove the shift that would produce the best change.

6. Remove all potentially superfluous shifts from the current state of GENET, forcing variables to choose different shifts.

Superfluous shifts are not alway present and in the latter stages of the search there are usually only 1 or 2, if any. Therefore, using a great deal of computing power and extra mechanisms to solve this problem is deemed to be fruitless. Therefore, the option decided on was strategy 6 and this was implemented by just tagging superfluous shifts to not be used in the next cycle, so that other shifts are used to cover the work. No attempt is made to ascertain whether two superfluous shifts would become non-superfluous if one were removed.

Table 6.3, giving results for the same problems as in Table 6.1, shows that adding a component to remove superfluous shifts at local minima does improve the solutions over just using the basic search, in several cases. We will see below that the problem of superfluous shifts disappears as we introduce general mechanisms to reduce the number

of shifts.

6.5 Optimisation

GENET has been used to solve optimisation problems and this has been described in sections 3.6.5 and 3.6.5.2. In the driver scheduling problem the most important criterion to optimise, the number of shifts, is a global criterion and so costs cannot be set on the labels initially. The cost of a label cannot be worked out locally because knowledge of the states of other variables is needed. A well known problem with a similar optimisation criterion is the Radio Frequency Assignment problem, when it is required to minimise the number of frequencies used. The difficulty in this type of minimisation is that to remove a frequency all the transmitters that are assigned to that frequency need to change state, which may require several independent moves. As mentioned in Section 3.6.5.2 this problem was solved by initially using only the minimum number of frequencies needed to cover all channels. This produces many constraint violations, but because the domains of most of the variables are the same, very few frequencies need be used, possibly only one. GENET then adds frequencies to reduce the conflicts and the number of frequencies added is thereby kept low.

Unfortunately, in the driver scheduling problem we cannot have one shift that would cover all the pieces, and choosing a minimal set of initial shifts covering all the pieces of work amounts to solving the problem. We can start with all the pieces uncovered, which is similar, although an uncovered piece is in conflict with everything, whereas one frequency will not be. By including a virtual shift as a value in the domain of all pieces of work, corresponding to the piece being left uncovered, we can start the process with this virtual shift chosen for all variables. We add a single constraint to penalise the use of the virtual shift and so GENET will add shifts to remove it. There is only one virtual shift which is heavily penalised. The risk of not finding a solution is low and all under-cover is normally removed in the first few cycles. The first entry (uncov) in Table 6.4 shows that the numbers of shifts in the solutions produced are less than or equal to the numbers of shifts produced

using a random start (the basic model of Table 6.1), but the improvement is small.

In order to find solutions with the same number of drivers as TRACS II, we need to address the difficulty that there are few opportunities to remove whole shifts. We can either put more effort into looking for global moves which will remove shifts by considering the solution as a whole or introduce a *bias* into GENET's local moves which will hopefully allow a sequence of local moves to lead to the removal of a shift. One possible way of doing this is by progressively penalising shifts that are not assigned to all their pieces of work, thus dissuading individual variables from using such a shift. This should guide the search to states where only one variable is using this shift and so to a position where it could be removed with a single change. To do this we add a term $P_{opt:<i,j>}$ to the input that a label node $< i, j >$ gets:

$$P_{opt:<i,j>} = -I + I_j - L_{<i,j>} \tag{6.1}$$

where I is the number of variables, I_j is the number of variables assigned to value j and $L_{<i,j>}$ is a number that starts at zero and increases by one every time that label node $< i, j >$ is *on* at a local minimum. The number of variables is obviously constant and is put in to ensure that the optimisation term is always inhibitory. The more variables there are which choose the value, the less inhibitory the term is.

A minor adaptation of this optimisation technique has been tried. The difference between it and the original technique is that shifts that are assigned to all the pieces they cover are not penalised. So there is no large negative influence of the constant value of the number of variables (I). This puts a large bias on shifts that are chosen by all their pieces. This was introduced to stabilise the search. By not penalising shifts that are chosen by all their pieces of work it is much more likely that these shifts will be retained.

Using the optimisation technique described above, Table 6.4 shows that in the overall averages there is a decrease in the number of shifts compared to the basic model. In comparison with the basic model of Table 6.1, it tends to produce slightly better solutions

Instance	t1	r1	r1a	r2	t2	r3	c1	c1a	r4	av
pieces	24	53	53	54	125	160	186	186	203	n/a
initial shifts	77	2503	4273	3001	3015	19091	3829	7543	2484	n/a
size	271	12k	21k	13k	17k	214k	27k	53k	17k	n/a
opt # shifts	7	11	11	14	19	16	26	26	25	17.22
uncov:										
av. # shifts	7.60	15.0	17.6	18.8	23.7	22.0	34.4	36.2	32.9	23.1
av. time (secs)	0.06	0.17	0.13	0.07	0.48	6.38	1.43	1.74	0.91	1.26
best # shifts	7	14	16	17	23	20	32	35	32	21.8
time (secs)	0.04	0.27	0.72	0.04	0.26	3.26	1.03	1.33	1.42	0.93
optl:										
av. # shifts	7.70	15.0	15.9	17.6	24.4	21.5	35.9	37.5	33.6	23.2
av. time (secs)	0.03	0.32	0.24	0.23	1.19	4.95	2.23	2.25	2.10	1.50
best # shifts	7	14	15	16	23	19	34	36	32	21.8
time (secs)	0.00	0.16	0.77	0.06	0.50	3.43	1.25	1.89	1.60	1.07
optl+nf:										
av. # shifts	7.70	15.0	16.0	18.5	23.6	21.9	34.7	36.1	32.8	22.9
av. time (secs)	0.00	0.08	0.05	0.02	0.35	1.59	0.42	0.49	0.65	0.41
best # shifts	7	14	14	18	22	20	34	35	32	21.8
time (secs)	0.00	0.06	0.09	0.01	0.08	1.54	0.20	0.32	0.19	0.28

Table 6.4: Results of using a technique to optimise the number of shifts used.
uncov = all the pieces of work start off uncovered
optl = the optimisation technique
nf = do not penalise shift that cover all their pieces

on smaller problems and worse on larger problems. This shows how difficult it is to remove a shift using a sequence of moves. A further problem is that to reduce the overall number of shifts a new shift may need to be introduced which fits better than an existing shift. In the optimisation scheme, shifts that are not used, i.e. no associated label node is *on*, will be heavily penalised and so are unlikely to be introduced, whether it is useful to do so or not. Below, we will retain the optimisation technique and investigate other ways to solve the two problems stated above.

The results in Table 6.4 also show that including the option to not penalise shifts that cover all their pieces only makes a minor difference by slightly decreasing the overall average number of shifts. This option will be further investigated in the final version of the search process.

6.5.1 Improved starting solution

By using a random initial solution to start the search, a large number of shifts is used. Since removing shifts is something that GENET finds difficult, a method was used to improve the quality of the initial solution. A simple greedy algorithm was used to create the initial solution. The algorithm starts with the earliest piece of work in the bus schedule. It then picks the shift to cover it that covers the largest number of other pieces of work. Then it continues picking uncovered work in chronological order and choosing the shift which covers the most uncovered pieces of work until all pieces of work are covered by at least one shift. The starting value (shift) assigned to each variable (piece) is then chosen randomly from the shifts that could cover it in this initial solution.

Table 6.5 shows the number of shifts used in the initial solution and the solutions found by GENET. In comparison with earlier results GENET improves the best found solution on several of the problems (**c1**, **t2**, **r2**). However, the best solution found is often no better than the initial solution. In the following sections, when testing new heuristics and adaptations of GENET, we will investigate combining the new features with the greedy initial solution.

Instance	t1	r1	r1a	r2	t2	r3	c1	c1a	r4	av
pieces	24	53	53	54	125	160	186	186	203	n/a
initial shifts	77	2503	4273	3001	3015	19091	3829	7543	2484	n/a
size	271	12k	21k	13k	17k	214k	27k	53k	17k	n/a
opt # shifts	7	11	11	14	19	16	26	26	25	17.22
Initsol	7	13	14	16	22	19	32	33	30	20.7
optl+init:										
av. # shifts	7.00	13.0	14.0	15.9	20.8	19.1	32.0	32.7	30.0	20.5
av. time (secs)	0.00	0.02	0.01	0.02	0.02	0.37	0.04	0.10	0.02	0.07
best # shifts	7	13	14	15	20	19	32	32	30	20.2
time (secs)	0.00	0.00	0.00	0.05	0.01	0.23	0.00	0.14	0.00	0.05

Table 6.5: Results using a greedy heuristic to construct an initial solution as opposed to a random starting solution.
Initsol = the number of shifts of the initial solution produced by the greedy heuristic
init = use initial solution produced by greedy heuristic
optl = optimisation with learning

6.5.2 Removing whole shifts

In this section we examine a way of using global moves to reduce the number of shifts. The idea is to take two shifts in the current solution and replace them with one shift. This would be very hard to do if there were little or no over-cover because it is then unlikely that the union of the pieces of work that two shifts cover is identical to the pieces of work covered by another shift. However, if there is enough over-cover it can be possible to find a shift that covers the work that two shifts cover uniquely between them. This is a good way of rapidly reducing the number of shifts and leaving the solution with little over-cover. We can combine these global moves with local moves and so let the local moves "fine tune" the solution. This is done by allowing the algorithm to work as normal until it reaches a local minimum, at which point it searches for two shifts that are in use that can be replaced by one shift. The results are shown in Table 6.6. The solutions are an improvement on the approach with the optimisation technique. However, it will be seen in later sections that we can do better using only local moves.

Instance	t1	r1	r1a	r2	t2	r3	c1	c1a	r4	av
pieces	24	53	53	54	125	160	186	186	203	n/a
initial shifts	77	2503	4273	3001	3015	19091	3829	7543	2484	n/a
size	271	12k	21k	13k	17k	214k	27k	53k	17k	n/a
opt # shifts	7	11	11	14	19	16	26	26	25	17.22
rep:										
av. # shifts	7.00	15.1	15.7	17.7	21.3	20.5	32.5	32.4	30.0	21.4
av. time (secs)	0.01	0.11	0.12	0.24	0.94	8.70	2.10	6.17	1.13	2.17
best # shifts	7	14	14	16	17	16	30	29	28	19.0
time (secs)	0.00	0.07	0.18	0.12	0.05	28.2	12.8	11.2	1.92	6.06
rep+init:										
av. # shifts	7.00	13.0	13.5	16.3	20.2	20.6	30.0	30.0	27.8	19.8
av. time (secs)	0.00	0.02	0.03	0.05	0.09	2.96	0.64	1.22	0.60	0.62
best # shifts	7	13	13	16	20	17	29	28	27	18.9
time (secs)	0.00	0.01	0.03	0.04	0.04	15.62	0.74	1.23	0.62	2.04

Table 6.6: Results showing the effect of using global moves to replace whole shifts
rep = try to replace two shifts with one
init = use initial solution produced by greedy heuristic

6.6 A less deforming learning model

In the model we originally developed, each constraint node has one associated weight. This weights the output to all the label nodes connected to that constraint. Each constraint node, is connected to all the label nodes representing two variables (see Figure 6.2). Through learning, the weight of the constraint node will increase every time any two label nodes connected to the constraint are *on* and are in conflict with each other. In the original version of GENET, there is a weight for every nogood pairing of label nodes, and this weight will only increase when this pairing of label nodes is *on* at a local minimum. With only one weight associated with a constraint the weight will increase more frequently than individual weights on nogood pairings. This is undesirable and so a compromise has been struck between having a weight for each nogood and only having one weight for each constraint. This compromise also has the added advantage of introducing bias into the model to reduce the number of shifts used.

To refine the learning method we have introduced more than one weight per constraint. We replace the single weight on the constraint (W) with a weight for each shift that is

in the domain of both of the variables (w_c where $c = 1, \ldots, C$ and C is the number of shifts in common) and a single weight (w_n) for all the shifts that are not common to both variables. Whether a label node is penalised or not is chosen in the same way as before, but how much it is penalised is chosen differently. Each label node is penalised by its associated weight. So the difference between the new constraint representation and the old one shown in Figure 6.2 is that we are using several weights instead of one. All the weights start at -1 as did W but the landscape for the two models become different through learning. With only one weight it increases every time the two associated variables are in conflict whereas in the new model only two of the weights increase (for example if the i and j were common shifts and both were *on* in a local minima only weights w_i and w_j would become more negative).

Figure 6.3 illustrates how using these extra weights works. There are now 3 weights stored: one each for shifts 3 and 8 as these are common to both pieces and one for the other shifts. All the weights start at -1 but through learning they can become different, in our example the weight for shift 8 has become -2 while the others remain at -1. In the example shown, if the constraint had only one weight and nodes `<A,3>` and `<B,2>` are *on*, nodes `<B,2>` and `<B,8>` would be penalised because they being *on* corresponds to over-cover. Similarly, `<A,8>` would be penalised because B is at the moment covered by shift 2 and so would be over-covered if shift 8 were also used. With multiple weights the same nodes are penalised but by different amounts, `<B,8>` by -2 because the weight linked with 8 is -2 and similarly `<A,3>` by -1.

Having extra weights has a twofold advantage over just having a single weight. Firstly, if there is only one weight the input of *all* penalised label nodes will become more negative by the same amount when the weight increases. Therefore, the associated variables are more likely to have nodes with the same input than if several different weights label nodes are used. This means that more moves are available if several different weights are used. The second advantage of using weights for shifts that are in common is that certain shifts will get more penalised, thus leading to their removal.

Instance	t1	r1	r1a	r2	t2	r3	c1	c1a	r4	av
pieces	24	53	53	54	125	160	186	186	203	n/a
initial shifts	77	2503	4273	3001	3015	19091	3829	7543	2484	n/a
size	271	12k	21k	13k	17k	214k	27k	53k	17k	n/a
opt # shifts	7	11	11	14	19	16	26	26	25	17.22
Initsol	7	13	14	16	22	19	32	33	30	20.7
mwt:										
av. # shifts	7.00	14.3	18.0	18.0	20.6	16.8	29.0	29.1	27.7	20.1
av. time (secs)	0.02	0.16	0.07	0.07	0.71	13.7	1.72	2.63	1.33	2.26
best # shifts	7	12	16	16	20	16	28	28	27	18.9
time (secs)	0.01	0.32	0.10	0.08	0.71	13.0	1.13	1.18	1.32	1.99
mwt+optl+init:										
av. # shifts	7.00	12.3	13.8	14.3	21.0	16.3	30.5	31.1	29.5	19.5
av. time (secs)	0.00	0.41	0.16	0.46	0.04	11.7	1.63	1.78	1.19	1.93
best # shifts	7	11	12	14	21	16	29	30	29	18.8
time (secs)	0.00	0.31	1.26	0.18	0.03	6.32	3.39	2.68	1.41	1.73
mwt+optl+rem:										
av. # shifts	7.00	12.1	14.0	14.0	22.2	16.3	30.7	30.1	29.2	19.5
av. time (secs)	0.03	0.55	0.47	0.56	1.00	13.8	4.40	4.94	2.49	3.14
best # shifts	7	11	12	14	22	16	29	29	28	18.7
time (secs)	0.00	0.85	2.27	0.17	0.36	6.56	6.44	2.84	2.01	2.39
mwt+optl+rep:										
av. # shifts	7.20	14.0	14.2	16.1	23.0	19.4	32.0	33.0	33.0	21.3
av. time (secs)	0.01	0.43	0.30	0.24	0.39	6.36	1.15	3.78	0.47	1.46
best # shifts	7	13	13	15	22	16	29	30	30	19.4
time (secs)	0.00	0.06	0.25	0.32	0.23	7.39	2.86	2.20	1.36	1.63
mwt+optl+nf:										
av. # shifts	7.00	12.8	14.8	16.1	20.7	16.5	29.1	29.2	27.9	19.4
av. time (secs)	0.25	2.24	0.93	1.32	4.74	17.4	8.34	10.41	8.02	5.96
best # shifts	7	12	12	14	20	16	28	28	27	18.2
time (secs)	0.03	1.45	1.60	3.12	5.15	15.14	5.28	17.7	4.55	6.00
mwt+optl:										
av. # shifts	7.00	11.8	14.1	14.2	22.3	16.3	30.5	30.4	29.5	19.6
av. time (secs)	0.03	0.46	0.24	0.63	0.93	9.90	2.00	2.37	1.37	1.99
best # shifts	7	11	11	14	21	16	29	29	28	18.4
time (secs)	0.00	0.52	0.45	0.19	0.48	6.03	3.52	2.19	1.20	1.62

Table 6.7: Using several weights for each constraint.
Initsol = the number of shifts of the initial solution produced by the greedy heuristic
optl = optimisation with learning
nf = do not penalise shifts that cover all their pieces
mwt = using more than one weight per constraint
init = use initial solution produced by greedy heuristic
rep = try to replace two shifts with one

Table 6.7 shows the results of using several weights to represent each constraint combined with the strategies considered earlier. There is a considerable improvement over the initial results shown in Table 6.1 and in many cases the best solutions found have the same number of shifts as in the TRACS II solution. It is no longer worthwhile to use the greedy heuristic to build a starting solution; a random initial solution does just as well. Hence the credit for the quality of the final solution is due entirely to the search algorithm. Furthermore, removing superfluous shifts now makes hardly any difference to the quality of the solutions. Equally the replace heuristic that removed whole shifts at local minima is no longer of use. The only alteration that does have a positive influence on the process of using several weights is not penalising shifts that are chosen by all the pieces of work that they can cover. By doing this one fewer shift is used in the best solution obtained in four of the larger problems. However, the best solution obtained for **r1** and **r1a** now contain one mores shift than the result produced by TRACS II.

By examining what the algorithm is doing during the search we can see how the improvement occurs. In its first two columns, Table 6.8 shows the average number of times a shift

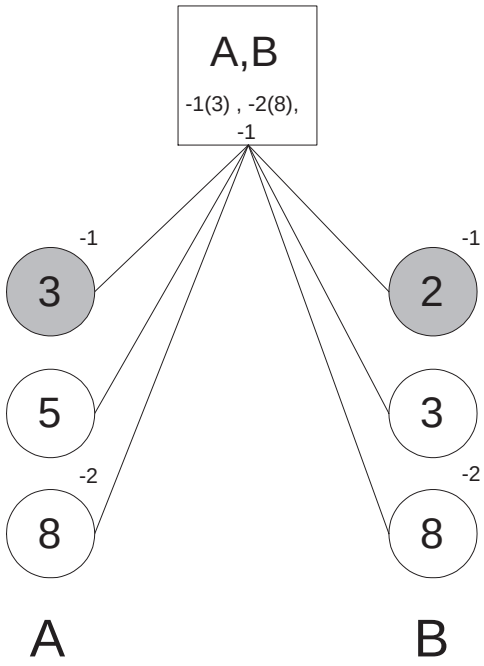

Figure 6.3: Set partitioning constraint node in GENET with more weight values

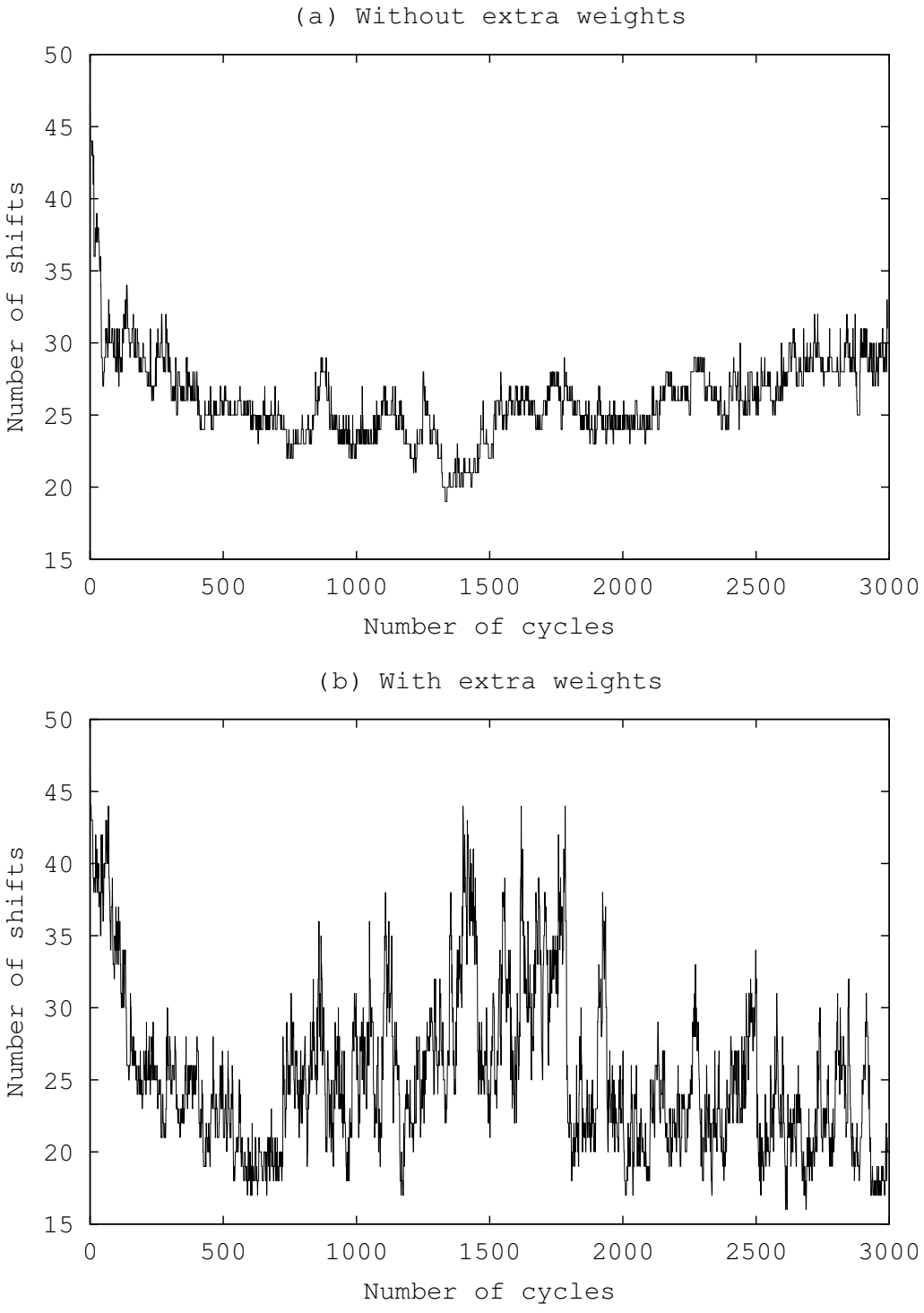

Figure 6.4: Number of shifts in the solution at each cycle of the search.

Instance	Revisits		Changes per cycle		% Local min	
	1 wt	> 1 wt	1 wt	> 1 wt	1 wt	> 1 wt
t1	3.32	1.86	1.63	2.51	48.63	40.23
r1	1.53	1.97	2.28	3.56	44.94	38.58
r1a	1.34	1.57	2.63	3.12	43.49	39.33
r2	1.27	1.71	2.63	3.32	43.78	39.66
t2	3.44	2.73	4.63	7.08	40.67	30.52
r3	3.62	3.04	7.80	16.80	31.13	22.80
c1	4.00	3.29	5.38	9.20	38.92	27.73
c1a	3.91	3.02	6.13	9.69	38.18	27.47
r4	3.77	3.29	5.33	9.41	35.61	27.34
av	2.91	2.50	4.27	7.18	40.59	32.63

Table 6.8: Comparison between one weight and multiple weights for each constraint. *Revisits* is the average number of times a shift is removed and later reinstated. *Changes per cycle* is the average number of clusters that change the label node that is *on* per cycle. *% Local minima* is the percentage of moves that are in a local minimum.

is removed and reinstated in GENET until the best solution is found. The table gives results for the 9 problems and the overall average. We can see that using more than one weight per constraint decreases, on average, the number of times a shift is revisited. The next two columns show the average number of variable clusters that change the label node that is *on* per cycle. In every case using more than one weight increases the number of changes and so does more searching on each cycle. Finally the last two columns show the percentage of moves that ended in a local minimum: the proportion of these unproductive moves is higher in every case when only one weight is used per constraint. Figure 6.4 compares in detail the search process with and without extra weights for a particular instance. Using the extra weights allows the search to change more at each cycle than when only one weight is used. It allows the search to move between states with few shifts even though it may have to temporarily add shifts to get between these states.

6.7 Summary and conclusion

Several adaptations to GENET have been made to try to reduce the number of shifts used. These consist of: introducing a bias in the local moves; making global moves; and

starting the search from an improved state. The most successful method was one of the attempts to include a bias in the local moves, by using multiple weights for each constraint. This made several of the other techniques obsolete. The reason for the success has been explained and evidence is given by examining the search process. This is interesting as using the biasing method enhances performance without adding additional mechanics to the search process, just adapting the constraints to improve the existing learning process.

As with the previous Chapter the research done here is domain specific however lessons may be learned for solving other practical and generals problems using GENET. These aspects are the following:

1. Optimisation. This research combined with that in [10, 9] shows the poor results obtained when trying to optimise a global optimisation criterion (number of drivers or number of frequencies) using the type of optimisation term used in Section 3.6.5 and 6.5. We have shown that the difficulty lies in having to make a succession of local moves to make a difference to the optimisation criterion. We have also introduced a method which allows for this type of sequence of local moves to improve the solution(Section 6.6).

2. Less deforming model. The original work on GENET for adding non-binary constraints discussed in Section 3.6.3 used only one weight per constraint node. However, we have shown in Section 6.6 that using more weights can make a significant difference to solution quality.

3. Sideways moves. How to decide whether sideways moves should be allowed when solving a particular problem using local search is still an open question. However, this research has put forward the idea that with hard, highly structured problems sideways moves should not be used. A full analysis of varying structure and its effect on the solution quality when using sideways moves is beyond the scope of the thesis but is an area for investigation.

Further work could be carried out in several areas. Using a relaxed linear programming

solution to the set partitioning problem greatly increased the performance of the systematic approach described in the previous chapter. Further, the only successful local search approach to large set/covering partitioning problems for driver scheduling [67] depends greatly on this LP solution. Therefore, examining how the LP solution could be incorporated into GENET may be very productive. The mechanics of using the LP solution need to be researched but there is a positive indication that it may work very well. This is because the LP solution and the GENET model have similarities. The assignment of a piece of work to a shift in GENET is similar to choosing a fraction of the shift to cover it in the LP solution. So translating the LP solution into a state in GENET would be a relatively easy task.

Another area for future work would be to expand the algorithm to tackle some of the further restrictions that can be imposed by bus companies. The driver scheduling problem sometimes has side constraints and features that are hard to express in a pure set partitioning formulation. Examination of these to see if the expressive power of constraint satisfaction can model these better than ILP could be very useful. This has been discussed in Section 5.8 and is further discussed in Section 7.3.

Lastly, a more general area for further work is to do with how GENET uses weights as we discussed in Item 2 above. The research dealt with the high memory requirements generated by representing the problem using GENET's original binary constraints. The problem was represented using an adaptation of the non-binary constraints developed for GENET to be used as binary constraints. It was found that using this type of constraint low quality results were produced, with large numbers of unnecessary drivers in the schedule. The possible reasons for this were examined by extracting information on the search. Using multiple weights instead of single weights improved the results greatly and the examination of the search gave possible reasons for this. An area open to research is whether in non-binary CSPs GENET should have each constraint with a single or multiple weight. A study of a range of problems with non-binary constraints, extracting the same search information, may shed light on this issue. So although this type of research is beyond the scope of this thesis, it has provided a direction for such research.

Chapter 7

Conclusions

7.1 Summary

The driver scheduling problem and its commercial importance has been presented. The current methods for driver scheduling have been described and their shortfalls expressed. The fact that it is sometimes hard to adapt methods between organisations and that provably optimal solutions to practical problems are not obtainable has been discussed. The problem is tightly restricted and heuristic methods have found it hard to produce good results for it. On the other hand, mixtures of heuristics and mathematical programming have been very successful, although even these have their flaws, which have stimulated investigation of other approaches.

This thesis has investigated two methods that use constraint satisfaction for modelling and solving the bus driver scheduling problem. These methods start from a predefined set of shifts, and from this they select shifts to produce a schedule. This tactic has been chosen over producing shifts as the schedule is built up because it allows the solver to be

generic and more independent of individual companies' regulations.

These methods have achieved success in solving small driver scheduling problems from different companies with varying regulations. However, the mathematical programming system TRACS II [37, 66, 125] can solve much larger problems. It is unsurprising that the new methods cannot compete, as there has been over 30 years of research invested in the TRACS II system. However, our results are encouraging, and indicate directions for further research.

7.2 Comparison between methods

The two new methods described in Chapters 5 and 6 are different in many ways. The first approach employs systematic search, whereas the second is a local search method developed from GENET [121, 110]. This means that in theory the systematic approach will, given enough time, produce an optimal solution but the local search method may never find the optimum. However, in practical terms, the problem is hard to solve and time is limited, so the systematic approach may also not find an optimal solution. Furthermore, as the set of possible shifts that are to be selected from is heuristically generated, shifts that are crucial to produce an optimal solution may not be contained in the set, leading to no optimal solution being obtained.

In examining results, the two methods cannot be directly compared. The two approaches tackle slightly different problems. GENET would accept set covering problems that the constraint programming approach will not. Moreover, the timings of runs cannot be compared as explained in Section 6.3.

For our test problems TRACS II produces the optimal number of drivers that can be achieved by selecting from the generated set of shifts. However, it is possible (although highly unlikely for problems of this size) that if we were to select from the set of all possible shifts, solutions with fewer drivers would exist. So therefore we will call a solution with the

same number of drivers as TRACS II a pseudo-optimal solution. It is worth noting in the research done in this thesis we do not consider associated costs of shifts, but TRACS II does and attempts to reduce them. Therefore, the pseudo-optimal solutions we speak of may in practice not be as good as the TRACS II solutions. In every one of the test cases, the systematic approach produced a pseudo-optimal solution. However, the GENET adaptation failed to produce the same number of drivers in four cases. Some factors may account for this difference in solution quality. In terms of the final version, GENET deals with the problem more as a general set partitioning problem than the systematic approach. GENET takes no advantage of the structure of the problem. On the other hand, the systematic constraint programming approach uses the solution to the LP given by relaxing the integrality conditions to guide the search and uses the structure of the bus schedule in the form of the relief opportunities to effectively reduce the size of the problem. It is believed that using the LP solution in some role within GENET will improve performance greatly. Without the use of the relaxed LP solution, the systematic approach could only find a pseudo-optimal solution on a trivially small problem instance. However, GENET has found a pseudo-optimal solution for the test problem with the largest number of potential shifts. Therefore, GENET may have the greater potential of the two.

An advantage GENET has over the systematic complete search method is that it will always find a solution of some quality. In the four cases in which it could not find a pseudo-optimal solution, the best solutions it found were only one or two shifts away from the TRACS II solution. Furthermore, GENET can handle set covering problems and so if there is no set partitioning solution it can still find a solution.

In this thesis when we compare the three search methods, mathematical programming, CP and Local search we can compare not just the first basic algorithms developed but also the comparison of how each approach can be adapted and improved. The mathematical approach has been developed over a long time and has been improved greatly with heuristics and improvements in its search technique. The CP approach has: examined modelling issues; used implied constraints, both mathematical and heuristic; used value and variable ordering; and used domain specific knowledge to enhance these. The local

search method explores several of the issues important to this type of search: escaping local minima; sideways moves; several techniques for optimisation including different moves operators; different starting solutions; and adapting the constraints. As stated above the GENET model did not incorporate as much domain specific knowledge as the constraint programming approach. Part of the reason for this is that in adapting and improving the GENET model the details of these improvements are often down to intuitive development from empirical evidence rather than the logical improvements possible with the CP approach. For example it is clear that a good value guide is useful to the CP approach but a good initial solution in the final GENET approach did not give improvements in the best solutions found.

7.3 Further work

Apart from the further developments that could be done individually to the two algorithms described in this thesis, there is also further research applicable to both, and alternative areas that do not directly relate to either algorithm, but to using constraint satisfaction in general for driver scheduling.

The individual areas of research for each algorithm are outlined in the conclusions of the relevant chapter. The following will summarise these. The systematic approach could benefit from further development in the implementation of the constraints to improve their time and space complexity. GENET could be advanced greatly by incorporating use of the structure of the problem and of the LP solution.

There are issues that could be explored possibly in extensions of both of these systems. Examining how regulations could be modelled in the constraint satisfaction framework would be of great value. For example, it would be useful to be able to model the frequent requirement that there is a maximum number of split shifts allowed in the schedule (see Section 4.3.2.8). This could be done simply in the systematic approach by having a variable for each split shift. These would have a binary domain (0,1) and be constrained

to have a 1 if the split shift was in use. A constraint would ensure that at most n of these variables would be permitted to have a value 1 at any single time, where n is the maximum allowed number of split shifts. How this would affect the quality the performance of the algorithm would be something to be tested. Further restrictions that are hard to model in the ILP approach are windows of relief opportunities and multi-depots (see Section 4.3.2.8). Windows of relief opportunity would be difficult to represent in any set partitioning/covering formulation, as such formulations deal with specific hand-over times. However, constraint satisfaction may provide the key. The reason for this are outlined in Section 5.8.

The problems used in this thesis for testing the algorithms produced were submitted to and will appear in the constraint satisfaction benchmarking library CSPlib [45]. This will allow other researchers access to the problems, so they can either develop new algorithms or perhaps test algorithms developed for air-crew scheduling set partitioning problems on driver scheduling problems. It would be of benefit to researchers to study the driver scheduling problem purely as a CSP. They might investigate how this CSP relates to randomly generated CSPs and to other practical problems formulated as CSPs. There are several aspects that can be investigated, and each may prove useful. One measure of the problem would be the *constrainedness* [42] which measures how restrictive the constraints of the problems are on the possible assignments. This would be useful because there have been studies on the constrainedness of problems and how this can be used in search [43]. Kwan [65] did a cursory examination of the number of solutions with the pseudo-optimal number of shifts. A more in-depth study could be conducted which could prove interesting in the light of such studies as Clark *et al* [16], which examined how local search is affected by the number of feasible solutions present in the search space. Walsh [120] examined how structure might affect search. The set partitioning problem is structured so that piece variables that represent consecutive pieces of work on the same bus are highly likely to have constraints between them. Variables representing pieces of work several hours apart are less likely to have constraints between them. This can affect which ordering is the best to use and we have seen a comparison of a dynamic ordering with a natural ordering,

as discussed in Section 5.9. Research into these issues would benefit the CSP community and may also provide knowledge on how to improve the constraint satisfaction approaches for driver scheduling. It would also be of interest to see how these measures would differ between air-crew, bus and train driver scheduling set partitioning problems.

7.4 Scope of research

Although the research in this thesis is domain specific there are areas of general use to the research community. Some of these have been highlighted in Sections 5.9 and 6.7. When reviewing the thesis each Section has its own scope, these can be categorised as:

1. Those only useful to the driver scheduling problem. These are the extended model in Section 5.5, superfluous shifts in Section 6.4 and removing whole shifts in Section 6.5.2.

2. Those useful to the set partitioning problem. These are the second model in Section 5.2.2 and the reductions in Section 5.4.

3. Those useful to applying GENET to general problems. When to apply sideways moves discussed in 6.7, analysis of search in Section 6.6 and the less deforming model also in Section 6.6.

4. Those useful for practical problems in general. These are fully detailed in Sections 5.9 and 6.7.

7.5 Achievements of the research

The research has allowed the comparison of three different search methods. This study has been carried out on only one type of problem. However, it has been a comprehensive study in that it explored many features of each of the techniques. So not only a basic model has been tried but many aspects of each type of search have been investigated.

This is different from the comparisons given in [22, 87] where only rudimentary models and search techniques where used.

The first stage of the research successfully extracted experience from the existing mathematical programming method for driver scheduling, TRACS II, and incorporated it into the constraint programming system to greatly improve the quality of solutions produced by the system. This new approach produces solutions for real driver scheduling problems. It has been shown to solve problems from different bus companies with different regulations, whereas for most of the recent modern heuristic approaches results have only been given for one company [12, 22, 129] . The size of these problems is much greater than pure CP approaches could solve. It also tested implied constraints (mathematical reductions, see Section 5.4) on the set partitioning problem which to the knowledge of the author has never been tried before. The work also highlights several aspects that may be of use in modelling other practical constraint satisfaction problems, as described in Section 5.9.

Local search processes have found the driver scheduling problem very hard. The solution space is rife with local minima and these swamp the global minima. Also the optimisation criterion, minimising the number of shifts, is difficult to tackle with the type of local search method GENET uses, as usually a succession of local moves need to be made to make an improvement. With the adaptations made to GENET, it has for several problems found pseudo-optimal solutions. It also demonstrated the examination of the search process and showed how these adaptations actual worked to improve the search. These adaptations and how they were examined may be of interest to those using GENET on similar problem areas. The adaptations are explained in Section 6.7 and guidelines for their general applicability are given.

A basic understanding of how constraint satisfaction can be used in driver scheduling has been developed and demonstrated this can be extended in future studies.

Bibliography

[1] R. Beale and T. Jackson. *Neural Computing: an Introduction*. IOP Publishing Ltd, 1992.

[2] J. E. Beasley. OR-Library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[3] J. C. Beck, A. J. Davenport, and M. S. Fox. Five pitfalls of empirical scheduling research. In *Principles and Practice of Constraint Programming - CP97*, pages 390–404. Springer, 1997.

[4] G. Bennington and K. Rebibio. Overview of RUCUS vehicle scheduling program(BLOCKS). In D. Bergmann and L. Bodin, editors, *Preprints: Workshop on Automated Techniques for Scheduling of Vehicle Operators for Urban Public Transportation Services*, 1975.

[5] C. Bessiere and M.-O. Cordier. Arc-consistency and arc-consistency again. In *Proc. of AAAI-93*, pages 108–113, 1993.

[6] C. Bessiere, E. C. Freuder, and J.-C. Regin. Using inference to reduce arc consistency computation. In *Proc. of IJCAI 95*, pages 592–598, 1995.

[7] J. Y. Blais and J. M. Rousseau. Overview of HASTUS current and future versions. In J. R. Daduna and A. Wren, editors, *Computer-aided Transport Scheduling*, pages 175–187. Springer-Verlag, 1988.

[8] A. Bockmayr and T. Kasper. Branch and infer: A unifying framework for integer and finite domain constraint programming. *INFORMS Journal on Computing*, 10:287–300, 1998.

[9] A. Bouju, J.F. Boyce, C. H. D. Dimitropoulous, and J. G. vom Scheidt, G. Taylor. Tabu search for the radio link frequency assignment problem. In *the International Conference on Digital Signal Processing*, 1995.

[10] J.F. Boyce, C.H. D. Dimitropoulous, G. vom Scheidt, and J. G. Taylor. GENET and tabu search for combinatorial optimization problems. In *World Congress on Neural Networks*. INNS press, 1995.

[11] Carlier, J. and Pinson, E. An Algorithm for Solving the Job-Shop Problem. *Management Science*, 35(2):164–176, 1989.

[12] L. Cavique, C. Rego, and I. Themido. Subgraph ejection chains and tabu search for the crew scheduling problem. *European Journal of Operational Research*, 50:608–616, 1999.

[13] B. Cha and K. Iwama. Performance test of local search algorithms using new types of random CNF formulas. In *Proc. of IJCAI 95*, pages 304–310, 1995.

[14] B. Cha and K. Iwama. Adding new clauses for faster local search. In *Proc. of AAAI-96*, pages 332–337. AAAI Press/MIT Press, 1996.

[15] P Charlier and H. Simonis. Abstract: A system for train crew scheduling. In *DIMACS Workshop on constraint programming and large scale discrete optimisation*, 1998.

[16] D. A. Clark, J. Frank, I. P. Gent, E. MacIntyre, N. Tomov, and T. Walsh. Local search and the number of solutions. In *Principles and Practice of Constraint Programming - CP96*, pages 119–133. Springer, 1996.

[17] M. D. J. Cox and E. P. K. Tsang. Application of GENET/GLS in future communications management. In *Advanced Software Applications using logic and Constraints*. CompulogNet, 1998.

[18] CPLEX. Using the CPLEX callable library, version 3.0. In *CPLEX Optimization, Inc*, 1994.

[19] S. D. Curtis, B. M. Smith, and A. Wren. Forming bus driver schedules using constraint programming. In *Practical Application of Constraint Technologies and Logic Programming Conference - PACLP99*, pages 239–254. The Practical Application Company Ltd, 1999.

[20] S. D. Curtis, B. M. Smith, and A. Wren. Constructing driver schedules using iterative repair. In *Practical Application of Constraint Technologies and Logic Programming Conference - PACLP2000*, page to appear. The Practical Application Company Ltd, 2000.

[21] J. R. Daduna and M. Mojsilovic. Computer-aided vehicle and duty scheduling using the HOT programme system. In A. Wren, editor, *Computer-Aided Transit Scheduling*, pages 133–146. Springer-Verlag, 1988.

[22] K. Darby-Dowman and J. Little. Properties of some combinatorial optimisation problems and their effect on the performance of integer programming and constraint logic programming. *INFORMS Journal on Computing*, 10:276–286, 1998.

[23] A. Davenport. *GENET Adaptation and Evaluation.* PhD thesis, Computer Science, University of Essex, 1997.

[24] A. Davenport and E. P. K. Tsang. Solving constraint satisfaction sequencing problems by iterative repair: an application to car sequencing. In *The Practical Application of Constraint Technologies and Logic Programming Conference - PACLP99*, pages 345–358. The Practical Application Company Ltd, 1999.

[25] A. Davenport, E. P. K. Tsang, C. J. Wang, and Z. Kangmin. GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proc. of AAAI-94*, pages 325–330. AAAI Press/MIT Press, 1994.

[26] M. Desrochers, J. Gilbert, M. Sauve, and F. Soumis. CREW-OPT: subproblem modeling in a column generation approach to urban crew scheduling. In M. Desrochers

and J. M. Rousseau, editors, *Computer-aided Transport Scheduling*, pages 395–406. Springer-Verlag, 1990.

[27] M. Desrochers and F. Soumis. CREW-OPT: crew scheduling by column generation. In J. R. Daduna and A. Wren, editors, *Computer-aided Transport Scheduling*, pages 83–90. Springer-Verlag, 1988.

[28] Y. Deville and P. Van Hentenryck. An efficient arc consistency algorithm for a class of CSP problems. In *Proc. of IJCAI 91*, pages 325–330, 1991.

[29] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving a cutting-stock problem in constraint logic programming. In R. Kowalski and K. Brown, editors, *Logic Programming*, 1988.

[30] M. Dorigo, V. Maniezzo, and A. Colorni. The ANT system: Optimisation by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics*, 25(12), 1995.

[31] H. El Sakkout. Modelling fleet assignment in a flexible environment. In *Practical Application of Constraint Technology - PACT96*, pages 27–39. The Practical Application Company Ltd, 1996.

[32] H. El Sakkout. *Improving backtrack search: three case studies of localized dynamic hybridization.* PhD thesis, IC-Parc, Imperial College of Science, Technology and Medicine, University of London, 1999.

[33] H. El Sakkout, E. T. Richards, and Wallace M G. Minimal perturbation in dynamic scheduling. In *Proc. of ECAI 98*, pages 47–51, 1998.

[34] J. C. Falkner and D. M. Ryan. Aspects of bus crew scheduling using a set partitioning model. In J. R. Daduna and A. Wren, editors, *Computer-aided Transport Scheduling*, pages 91–103. Springer-Verlag, 1988.

[35] J. C. Falkner and D. M. Ryan. EXPRESS: set partitioning for bus crew scheduling in Christchurch. In M. Desrochers and J. M. Rousseau, editors, *Computer-aided Transport Scheduling*, pages 359–378. Springer-Verlag, 1990.

[36] S. Fores. *Column Generation Approaches to Bus Driver Scheduling.* PhD thesis, School of Computer Studies, University of Leeds, 1996.

[37] S. Fores, L.G. Proll, and A. Wren. An improved ILP system for driver scheduling. In N.H.M. Wilson, editor, *Computer-Aided Scheduling of Public Transport*, pages 43–62. Springer, 1999.

[38] P. Forsyth and A. Wren. An ant system for driver scheduling. Technical Report 97.25, Computer Studies, University of Leeds, 1997.

[39] E. C. Freuder. Modeling: The final frontier. In *Practical Application of Constraint Technologies and Logic Programming Conference - PACLP99*, pages 15–22. The Practical Application Company Ltd, 1999.

[40] R. S. Garfinkel and G. L. Nemhauser. *Integer Programming.* Wiley-Interscience, 1972.

[41] P. A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proc. of ECAI 92*, pages 31–35, 1992.

[42] I. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Proc. of AAAI-96*, pages 246–252. AAAI Press/MIT Press, 1996.

[43] I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Principles and Practice of Constraint Programming - CP96*, pages 179–193. Springer, 1996.

[44] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proc. of AAAI-93*, pages 28–33. AAAI Press/MIT Press, 1993.

[45] I.P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical Report APES-09-1999, University of Strathclyde, 1999. Available from http://csplib.cs.strath.ac.uk/. A shorter version appears in: Principles and Practices of Constraint Programming - CP99.

[46] F. Glover and M. Laguna. *Tabu search*. Kluwer Academic Publishers, 1998.

[47] S. W. Golomb and L. D. Baumert. Backtracking programming. *Journal of the ACM*, 12:516–524, 1965.

[48] N. Guerinik and M. V. Caneghem. Solving crew scheduling problems by constraint programming. In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming - CP95*, pages 481–498. Springer, 1995.

[49] N. S. Hans. Constraint satisfaction problems. In C. T. Leondes, editor, *Optimization Techniques*, pages 209–248. Academic Press Ltd, 1998.

[50] J. Hao and R. Dorne. Empirical studies of heuristic local search for constraint solving. In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming - CP96*, pages 194–208. Springer, 1996.

[51] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[52] S. Helmut. The CHIP system and its applications. In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming - CP95*, pages 643–646. Springer, 1995.

[53] K. L. Hoffman and M. Padberg. Solving air crew-scheduling by branch-and-cut. Technical report, Geoge Mason University and New York University, 1992.

[54] J. Hoffstadt. Computerized vehicle and driver scheduling for the hamburger Hochbahn Aktiengesellschaft. In A. Wren, editor, *Computer Scheduling of Public Transport*, pages 35–52. North-Holland, 1981.

[55] J. N. Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1(1):33–42, 1995.

[56] J. N. Hooker, Ottosson G., T. S. Erlendur, and H-J. Kin. On intergrating constraint propogation and linear programming for combinatortial optimisation. In *Proc. of AAAI-99*, pages 136–141. AAAI Press/MIT Press, 1999.

[57] ILOG. *Solver reference manual*, 3.2 edition, 1996.

[58] ILOG. *Solver user manual*, 3.2 edition, 1996.

[59] Forrest J. J. and D. Goldfarb. Steepest edge simplex algorithms for linear programming. *Mathematical Programming*, 57:341–374, 1992.

[60] Wallace R. J. Analysis of heuristic methods for partial constraint satisfaction problems. In *Principles and Practice of Constaint Programming - CP96*, 1996.

[61] M. D. Johnston and H. M. Adorf. Learning in stochastic neural networks for constraint satisfaction problems. In *Proc. of the NASA conference on Space Telerobotics*, 1989.

[62] K. Kask and R. Dechter. GSAT and local consistency. In *Proc. of IJCAI 95*, pages 616–622, 1995.

[63] N. Keng and D. Y. Y. Yun. A planning/scheduling methodology for the constrained resource problem. In *Proc. of IJCAI 89*, pages 998–1003, 1989.

[64] S. Kirkpatrick, Gelatt C. D., and Vecchi M. P. Optimization by simulated annealing. *Science*, (220):671–680, 1983.

[65] A.S.K. Kwan. *Train Driver Scheduling*. PhD thesis, School of Computer Studies, University of Leeds, 1999.

[66] A.S.K. Kwan, R.S.K. Kwan, M.E. Parker, and A. Wren. Producing train driver schedules under differing operating strategies. In N.H.M. Wilson, editor, *Computer-aided Transport Scheduling*, pages 129–154, 1999.

[67] A.S.K. Kwan, R.S.K. Kwan, and A. Wren. Driver scheduling using genetic algorithms with embedded combinatorial traits. In N.H.M. Wilson, editor, *Computer-aided Transport Scheduling*, pages 81–102, 1999.

[68] Lau T. L. and Tsang E. P. K. Solving the processor configuration problem with a mutation-based genetic algorithm. *International Journal on Artificial Intelligence Tools - IJAIT97*, 6(4):567–585, 1997.

[69] C. J. Layfield, B. M. Smith, and A. Wren. Bus relief opportunity selection using constraint programming. In *Practical Application of Constraint Technologies and Logic Programming Conference - PACLP99*, pages 537–552. The Practical Application Company Ltd, 1999.

[70] J. H. M. Lee, P. J. Stuckey, V. W. L. Tam, and H. W. Won. Performance of a comprehensive and efficient constraint library using local search. In *11th Australian Joint Conference on Artificial Intelligence*, pages 191–202. Springer-Verlag, 1998.

[71] J.H.M. Lee, H. Leung, and H. Won. Towards a more efficient stochastic constraint solver. In *Principles and Practice of Constaint Programming - CP96*, pages 531–552, 1996.

[72] J.H.M. Lee, H. F. Leung, and H. W. Won. Extending GENET for non-binary constraint satisfaction problems. In *7th International Conference on Tools with Artificial Intelligence*, pages 338–342, 1995.

[73] J. Leinbach. Automatic local annealing. In D. S. Touretzky, editor, *Advances in Neural Information Processing systems 1*, pages 602–609. Morgan Kaufmann Publishers Inc., 1989.

[74] L. K. Luedtke. RUCUS II: A review of system capabilities. In J.-M. Rousseau, editor, *Computer Scheduling of Public Transport 2*, pages 61–116. North-Holland, 1985.

[75] L. D. Bodin M. O. Ball and J. Greenberg. Enhancements to the RUCUS II crew scheduling system. In J.-M. Rousseau, editor, *Computer Aided Scheduling of Public Transport 2*, pages 279–294. North-Holland, 1985.

[76] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 28:99–118, 1986.

[77] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[78] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–204, 1992.

[79] R. Mohr and T. C Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[80] P. Morris. The breakout method for escaping from local minima. In *Proc. of AAAI-93*, pages 40–45. AAAI Press/MIT Press, 1993.

[81] T. Müller. Solving set partitioning problems with constraint programming. In *PAPPACT98*, pages 313–332. The Practical Application Company Ltd, 1998.

[82] M. E. Parker and B. M. Smith. Two approaches to computer crew scheulding. In A. Wren, editor, *Proc. of the Second International Workshop on Computer-Aided Scheduling of Public Transport*, pages 193–222. North-Holland, 1981.

[83] L. Proll and B. M. Smith. Integer linear programming and constraint programming approaches to a template design problem. *INFORMS Journal on Computing*, 10:265–275, 1998.

[84] J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In J. Komorowski and Z. W. Ras, editors, *Methodologies for Intelligent Systems: Proc. of the 7th International Symposium ISMIS-93*, pages 350–361. Springer, 1993.

[85] J.F. Puget. A C++ implementation of CLP. In *Proc. of SPICIS 94*, 1994.

[86] J.F. Puget. A comparison between constraint programming and integer programming. In *Conference on Applied Mathematical Programming and Modelling (APMOD95)*, 1995.

[87] R. Rodosek, M.G. Wallace, and T. Hajian. A new approach to integrate mixed integer programming. *CP96 workshop on Constraint Programming Applications: An Inventory and Taxonomy*, 1996.

[88] J-M. Rousseau and M. Desrochers. Results obtained with CREW-OPT, a coloumn generation method for transit crew scheduling. In J. R. Dauna, I. Branco, and Paixao, editors, *Computer-aided Transport Scheduling*, pages 349–358. Springer-Verlag, 1995.

[89] D.M. Ryan and B.A. Foster. An integer programming approach to scheduling. In A. Wren, editor, *Computer Scheduling of Public Transport*, pages 269–280. North-Holland Publishing Company, 1981.

[90] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction problems. In A. Cohn, editor, *Proc. of ECAI 94*, pages 125–129. John Wiley & Sons, Ltd, 1994.

[91] M. Sabin and E. C. Freuder. Automated formulation of constraint satisfaction problems. In *Proc. of AAAI-96*, page 1407. AAAI Press/MIT Press, 1996.

[92] B. Selman and H. A. Krautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In R. Bajcsy, editor, *IJCN-93*, pages 290–295. Morgan Kaufmann Publishers Inc, 1993.

[93] B. Selman and H. A. Krautz. An empirical study of greedy local search for satisfiability testing. In *Proc. of AAAI-93*, pages 46–51. AAAI Press/MIT Press, 1993.

[94] B. Selman and H. A. Krautz. Noise strategies for improving local search. In *Proc. of AAAI-94*, pages 337–343. AAAI Press/MIT Press, 1994.

[95] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiablility problems. In *Proc. of AAAI-92*, pages 440–446. AAAI Press/MIT Press, 1992.

[96] B. M. Smith. Succeed-first or fail-first: A case study in variable and value ordering heuristics. In *Proc. of PACT97*, pages 321–330. The Practical Application Company Ltd, 1997.

[97] B. M. Smith, S. C. Brailsford, P. M. Hubbard, and H.P. Williams. The progressive party problem: Integer programming and constraint programming compared. In

U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming - CP95*, pages 36–52. Springer, 1995.

[98] B. M. Smith and S. A. Grant. Trying harder to fail first. In Henri Prade, editor, *Proc. of ECAI 98*, pages 249–253, 1998.

[99] B.M. Smith. *Bus Crew Scheduling Using Mathematical Programming.* PhD thesis, School of Computer Studies, University of Leeds, 1986.

[100] B.M. Smith and A. Wren. A bus crew scheduling system using set covering formulation. *Transpn.Res.*, (22A):97–108, 1988.

[101] M. Sqalli and E. C. Freuder. Inference-based constraint satisfaction supports explanation. In *Proc. of AAAI-96*, pages 318–325. AAAI Press/MIT Press, 1996.

[102] P. Stuckey and V. Tam. Extending EGENET with lazy constraint consistency. In *IEEE 9th International Conference on Tools with AI*, 1997.

[103] G. A. Tagliarini and E. W. Page. Solving constraint satisfaction problems with neural networks. In *Proc. of the International Joint Conference on Neural Networks*, 1987.

[104] G. A. Tagliarini and E. W. Page. Learning in systematically designed networks. In *Proc. of the International Joint Conference on Neural Networks*, 1989.

[105] J. Thornton and A. Sattar. Using arc weights to improve iterative repair. In *Proc. of AAAI-98*, pages 367–372. AAAI Press/MIT Press, 1998.

[106] E. P. K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[107] E. P. K. Tsang. No more "partial" and "full looking ahead". *Artificial Intelligence*, 98:351–361, 1998.

[108] E. P. K. Tsang, Wang C. J., A. Davenport, C. Voudouris, and T. L. Lau. A family of stochastic methods for constraint satisfaction. In *Practical Application of Constraint Technologies and Logic Programming Conference - PACLP99*, pages 359–385. The Practical Application Company Ltd, 1999.

[109] E. P. K. Tsang and C. Voudouris. Fast local search and guided local search and their application to british telecom's workforse scheduling problem. In *Practical Application of Constraint Technology - PACT96*, volume 20, pages 119–127, Amsterdam, 1997. Elsevier Science Publishers.

[110] E. P. K. Tsang and C. J. Wang. A generic neural network approach for constraint satisfaction problems. *Neural Network Applications*, pages 12–22, 1992.

[111] M. Völker and P. Schütze. Recent developments of the HOT system. In I. Branco J. R. Daduna and J. M. P. Paixáo, editors, *Computer-Aided Transit Scheduling*, pages 334–348. Springer-Verlag, 1995.

[112] C. Voudouris. *Guided Local Search for Combinatorial Optimisation Problems*. PhD thesis, Department of Computer Science, University of Essex, 1997.

[113] C. Voudouris and E. P. K. Tsang. The tunneling algorithm for partial CSPs and combinatorial optimization problems. Technical Report CSM-213, University of Essex, 1994.

[114] C. Voudouris and E. P. K. Tsang. Guided local search. Technical Report CSM-247, University of Essex, 1995.

[115] C. Voudouris and E. P. K. Tsang. Partial constraint satisfaction and guided local search. In *Practical Application of Constraint Technology - PACT96*, pages 337–356. The Practical Application Company Ltd, 1996.

[116] C. Voudouris and E. P. K. Tsang. Guided local search joins the elite in discrete optimisation. In *DIMACS Workshop on Constraint Programming and Large Scale Discrete Optimisation*, 1998.

[117] C. Voudouris and E. P. K. Tsang. Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research*, 113:469–499, 1999.

[118] M. Wallace, S. Novello, and J. Schimpf. ECLiPSe: a platform for constraint logic programming. Technical report, Imperial College, 1997.

[119] R. J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc-consistency in CSPs. In *Proc. of AAAI-93*, pages 239–245. AAAI Press/MIT Press, 1993.

[120] T. Walsh. Search in a small world. In *Proc. of IJCAI 99*, pages 1172–1178, 1999.

[121] C.J. Wang and E.P.K. Tsang. Solving constraint satisfaction problems using neural networks. In *IEE Second International Conference on Artificial Neural Networks*, pages 295–299, 1991.

[122] C.J. Wang and E.P.K. Tsang. A cascadable VLSI design for GENET. In *International Workshop on VLSI for Neural Networks and Artificial Intelligence*, 1992.

[123] W. P. Willers. *Improved Algorithms for Bus Crew Scheduling*. PhD thesis, School of Computer Studies, University of Leeds, 1995.

[124] J. H. Y. Wong and H. F. Leung. Solving fuzzy constraint satisfaction problems with fuzzy GENET. In *IEEE ICTAI98*, 1998.

[125] A. Wren and R. S. K. Kwan. Installing an urban transport scheduling system. *Journal of Scheduling*, 2:3–17, 1999.

[126] A. Wren and J-M. Rousseau. Bus driver scheduling -an overview. In J. R. Daduna, I. Branco, and J. M. P. Paixáo, editors, *Computer-aided Transit Scheduling*, pages 173–187. Springer-Verlag, 1995.

[127] A. Wren and D. O. Wren. A genetic algorithm for public transport scheudling. *Computers and Operations Research*, 22:101–110, 1995.

[128] N. Yugami, Y. Ohta, and H. Hara. Improving repair-based constraint satisfaction methods by value propagation. In *Proc. of AAAI-94*, pages 344–349. AAAI Press/MIT Press, 1994.

[129] T. H. Yunnes, A. V. Moura, and C. C. de Souza. Solving large scale crew scheduling problems with constraint programming and integer programming. Technical Report IC 99-19, Institute of Computing, UNICAMP, 1999.

Glossary

This is a glossary of the transport scheduling terms used in this thesis. Note that different transport companies may have different meanings for the words, described here are the meanings purely for this thesis.

depot: A centre of operation for a company. Normally a place where vehicles and crews are dispatched from at the start of their work period and returned to at the end of it.

flight leg: The equivalent to a **piece of work** in air crew scheduling

joinup: The time period between two **spells** of work that allows time to change buses but is not a **meal break**.

meal break: A rest break during a **shift** which must be of a certain length as specified by **union agreements**.

over-cover: When two or more drivers are on the same bus during a **piece of work**.

piece of work: An indivisible period of driving work, between two **relief opportunities**.

relief opportunities (RO): A **relief time** and **relief point** pairing to stipulate a specific time and place where drivers can change over.

relief point: Designated locations on bus routes where drivers may change over.

relief time: A time when a bus passes a **relief point**.

rotation: The equivalent to a **shift** in air crew scheduling

running board: A description of the work a bus does in a day.

shift/duty: The work a driver does in a day, normally consisting of two **stretches** of work seperated by a **meal break**.

spell: A continuous period of driving on one bus.

split shift: A type of shift where the driver has a much longer break in the middle of the shift than a normal shift.

stretch: One or more spells of work in a **shift**, each spell being on a different bus and separated by a **joinup**.

union agreements: Rules agreed between staff unions and the company concerning driving conditions.

window of relief opportunity: The time that a vehicle remains at a **relief point**, such as a bus station, where there is a choice of times to change the driver.